Fast, Parallel Tools for Genome-wide Analysis of Genomic Divergence

Tuva Kristine Thoresen

Master's Thesis August 2015

Fast, Parallel Tools for Genome-wide Analysis of Genomic Divergence

Tuva Kristine Thoresen

17th August 2015

Abstract

Comparative genomics is useful for finding the evolutionary relationships between different organisms. Regions of parallel evolution can be identified, and we can gain a better understanding of evolution and the development of different species. Tools for comparative genomics have previously been developed by Vederhus (2013). The tools, however, had some shortcomings, such as the usability and the long runtime of the tools. The aim of this thesis was to improve the tools for comparative genomics.

We have implemented two different methods for locating diverting regions in the genome of two populations, a Fisher's Exact Test (FET) and a Cluster Separation Score (CSS). For CSS, three different methods for calculating multidimensional scaling (MDS) were implemented. The tools were implemented as web tools on the Genomic HyperBrowser, using Python, Cython and C, and the code was parallelized with Pthreads. We have simplified the file format and given the user more choices in parameters, and created a tool for converting VCF files to our file format.

The changes in file format and the VCF convert tool have made the tool more usable for a broader range of applications, and by calculating the complete two-tailed FET and implementing three different methods for calculating MDS, we have achieved a more accurate result. We found that we were able to gain a large speedup of our tools, giving a dramatic decrease in run time. This should make it possible to run several analyses with different parameters in a short amount of time. The tools were able to reproduce the results found in previous studies, and produce good results on two additional data sets. The CSS was the most accurate method, achieving good results over a broad range of data sets, while the FET had more noise and is weaker on data sets with little genomic divergence.

The increased speed and user friendliness of the tools make it feasible to run these analyses on a larger scale than has previously been done. These tools should be able to meet the increased need for analysing large scale data sets in comparative genomics.

Acknowledgements

This study was carried out at the Biomedicial Informatics (BMI) research group at the Department of Informatics at the University of Oslo.

First, I would like to thank my main supervisor Professor Torbjørn Rognes for thorough reading of my thesis, help with the methods and parallel program, and a lot of patience and advice. I would also like to thank my supervisor Geir Kjetil Sandve for help with methods and information about the Genomic Hyper-Browser. Further, I would like to thank Sveinung Gundersen and Abdulrahman Azab from the BMI group, for their invaluable help with the HyperBrowser system, for merging in the newest version, installing tools and helping with correcting errors. I would also like to thank Bastiaan Star, at the Centre for Ecological and Evolutionary Synthesis at the Department of Biosciences at the University of Oslo, for his help with the Atlantic cod data and the VCF file format.

On a more personal note, I would like to thank my fellow students at the TekNat master, especially Ingrid Grønlie Guren, for help and tips, and all the girls at Verdande and my fellow group teachers, for making my time at IFI memorable. I would also like to thank my mother and father, for help and ideas and thorough reading of my thesis. Finally, I wish to thank my boyfriend Edvin, for his support and patience, and for taking more than his share of household duties.

Contents

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Research questions . . . 1
1.3 Overview of thesis . . . 2

2 Background **3**
2.1 Essential biological concepts . . . 3
2.2 The genomic basis of adaptive evolution in threespine sticklebacks 4
2.3 A long term evolution experiment with *Drosophila* . . . 5
2.4 The Genomic HyperBrowser . . . 6
2.4.1 HyperBrowser jobs and tracks . . . 6
2.4.2 Statistics . . . 7
2.5 A Tool for Genome-wide Analysis . . . 7
2.5.1 Tool implementation in HyperBrowser . . . 8
2.5.2 Issues with the current tool . . . 9
2.6 Performance metrics . . . 9
2.6.1 Runtime . . . 9
2.6.2 Overhead . . . 9
2.6.3 Speedup and efficiency . . . 10

3 Methods **11**
3.1 Cluster Separation Score (CSS) . . . 11
3.1.1 Multi-Dimensional Scaling (MDS) . . . 12
3.1.2 Classical MDS . . . 13
3.1.3 Iterative MDS . . . 14
3.1.4 Statistical significance . . . 16
3.2 Fisher's Exact Test (FET) . . . 17
3.2.1 Fisher's Exact Test algorithm . . . 18
3.3 Parallel programming systems . . . 20
3.3.1 Message Passing Interface (MPI) . . . 20

3.3.2 Pthreads 22
3.3.3 OpenMP 24
3.3.4 MapReduce 24
3.4 Parallel design models 25
3.4.1 The boss/worker model 25
3.4.2 The peer model 25
3.4.3 The pipeline model 25
3.5 Mapping of tasks in parallel systems 26
3.5.1 Overhead 26
3.5.2 Tasks and work division 26

4 Implementation 27
4.1 Implementation choices 27
4.1.1 Main structure of the tools 27
4.1.2 Methods 28
4.1.3 Languages and frameworks 28
4.1.4 Parallelization 29
4.2 Data structure 30
4.2.1 File format 31
4.2.2 A tool for converting VCF to GTrack 32
4.2.3 Sliding windows 32
4.3 Fisher's Exact Test 33
4.3.1 The web tool 33
4.3.2 The statistic 33
4.4 Cluster Separation Score 35
4.4.1 Distance metrics 37
4.4.2 MDS methods 38
4.4.3 CSS 39
4.4.4 Estimating significance 40
4.5 Parallel implementation 41
4.5.1 Pthreads implementation 41
4.6 Pseudo-random number generators (PRNGs) 43
4.6.1 A thread safe pseudo-random number generator 43
4.6.2 Uniform distribution of pseudo-random numbers 44
4.7 Integrating C code with Cython 44
4.7.1 Problems with integrating C code in a large scale Python system 45
4.8 Optimizing C code 46
4.8.1 Allocations 47
4.8.2 Library methods 47
4.8.3 Pseudo-random number generators 47
4.8.4 For-loops 47
4.8.5 Data types 47
4.8.6 Functions 48
4.9 Changes made to the HyperBrowser code 48

5 Results **49**
5.1 User interface 49
5.1.1 Fisher Exact Test SNP Tool 50
5.1.2 Filter Fisher Scores 50
5.1.3 Cluster Separation Score 50
5.1.4 Significant CSS Regions 51
5.1.5 Convert Stickleback SNPs to GTrack 51
5.1.6 Convert VCF To GTrack Tool 51
5.2 Speedup of serial C code vs Python code 51
5.2.1 Cluster Separation Score 52
5.2.2 Fisher's Exact Test 54
5.3 Memory usage 55
5.3.1 Cluster Separation Score 56
5.3.2 Fisher's Exact Test 57
5.4 Parallel C code 57
5.4.1 Variable number and size of tasks 58
5.4.2 Variable number of threads 60
5.5 Synthetic data set 65
5.5.1 Cluster Separation Score 65
5.5.2 Fisher's Exact Test 71
5.6 Three-spined stickleback data 77
5.6.1 Cluster Separation Score 77
5.6.2 Fisher's Exact Test 81
5.7 *Drosophila* data 86
5.7.1 Cluster Separation Score 86
5.8 Atlantic cod data 88
5.8.1 The two marine populations 89
5.8.2 The marine and coastal 2011 populations 91

6 Discussion and Conclusion **95**
6.1 Discussion of the usability of the tools 95
6.2 Discussion of speedup and code integration 96
6.3 Discussion of the results from analyses 97
6.3.1 Synthetic data 97
6.3.2 Three-spined stickleback data 98
6.3.3 *Drosophila* data 98
6.3.4 Atlantic cod data 99
6.3.5 Possible weaknesses in analyses 100
6.4 Weakness in implementation 101
6.5 Conclusion 101

7 Future Work **103**

Appendices **105**

Appendix A Example runs **107**
A.1 An example run with three-spined stickleback data 107
A.1.1 Converting the three-spined stickleback data file 107
A.1.2 The Fisher's Exact Test Tool 110
A.1.3 The Cluster Separation Score Tool 111
A.2 VCF Convert example run 114

Appendix B Analyses on the Genomic HyperBrowser **115**

Appendix C Detailed results **117**
C.1 Detailed results of parallel program 117
C.2 Detailed three-spined stickleback results 120
C.2.1 Cluster Separation Scorer 120
C.3 Detailed *Drosophila* results 132
C.4 Detailed Atlantic cod results 136
C.4.1 The two marine populations 136
C.4.2 The marine and coastal 2011 populations 136

Appendix D Source code **143**

References **145**

List of Figures

5.1 Variable task size: CSS 59
5.2 Variable task size: FET 60
5.3 Variable number of threads: CSS 61
5.4 Scaled speedup: CSS 62
5.5 Variable number of threads, ts 25: CSS 62
5.6 Scaled speedup, ts 25: CSS 63
5.7 Variable number of threads: FET 64
5.8 Scaled speedup: FET 64
5.9 Synthetic data: CSS with noise 0 67
5.10 Synthetic data: CSS with noise 0.5 68
5.11 Synthetic data: CSS with noise 0.8 69
5.12 Synthetic data: CSS with noise 0.9 70
5.13 Synthetic data: FET with noise 0 72
5.14 Synthetic data: FET with noise 0.5 73
5.15 Synthetic data: FET with noise 0.8 74
5.16 Synthetic data: FET with noise 0.9 75
5.17 Synthetic data: FET with noise 1 76
5.18 Stickleback chrIV EDA gene: CSS 78
5.19 Stickleback chrIV: CSS significant regions 79
5.20 Stickleback chrIV EDA gene: CSS significant regions 79
5.21 Stickleback chrXIX: CSS significant regions 80
5.22 Stickleback chrXIX detailed results: CSS 80
5.23 Stickleback chrIV: Filtered FET regions with Burke et al. limit . 82
5.24 Stickleback chrIV EDA gene: FET 83
5.25 Stickleback chrIV: Filtered FET regions 83
5.26 Stickleback chrIV EDA gene: Filtered FET regions 84
5.27 Stickleback chrXIX: Filtered FET regions 84
5.28 Stickleback chrXIX detailed results: FET 85
5.29 *Drosophila* chrX: CSS CMDS + SMACOF 87
5.30 *Drosophila* chrX: top scoring regions 88
5.31 Atlantic cod, marine populations: CSS CMDS 89
5.32 Atlantic cod, marine populations: LG01 for FET and CSS 90
5.33 Atlantic cod, marine populations: CSS - filtered regions in LG01 90
5.34 Atlantic cod, the 2011 populations: CSS CMDS 91

5.35 Atlantic cod, the 2011 populations: CSS SMACOF 92
5.36 Atlantic cod, the 2011 population: CSS - filtered regions in LG01 92
5.37 Atlantic cod, the 2011 populations: FET 93
5.38 Atlantic cod, the 2011 population: Filtered FET regions 94

A.1 The sticklebrowser 108
A.2 Converting to GTrack file 109
A.3 Fisher Exact Test SNP Tool 110
A.4 Filter Fisher Scores 112
A.5 Significant CSS Regions 112
A.6 Cluster Separation Score 113
A.7 Convert VCF To GTrack Tool 114

C.1 Stickleback chrI: CSS significant regions 120
C.2 Stickleback chrII: CSS significant regions 121
C.3 Stickleback chrIII: CSS significant regions 122
C.3 Stickleback chrIV: CSS significant regions 123
C.4 Stickleback chrVI: CSS significant regions 124
C.5 Stickleback chrVII: CSS significant regions 124
C.6 Stickleback chrVIII: CSS significant regions 125
C.7 Stickleback chrIX: CSS significant regions 127
C.8 Stickleback chrX: CSS significant regions 127
C.9 Stickleback chrXI: CSS significant regions 128
C.10 Stickleback chrXII: CSS significant regions 129
C.11 Stickleback chrXVI: CSS significant regions 129
C.12 Stickleback chrXVIII: CSS significant regions 130
C.13 Stickleback chrXX: CSS significant regions 131
C.14 Stickleback chrUn: CSS significant regions 131
C.15 *Drosophila*: CSS CMDS 132
C.16 *Drosophila*: CSS SMACOF 134
C.17 Atlantic cod, marine populations: CSS CMDS 137
C.18 Atlantic cod, the 2011 populations: CSS - filtered regions in LG02 138
C.19 Atlantic cod, the 2011 populations: CSS - filtered regions in LG07 139
C.20 Atlantic cod, the 2011 populations: FET 139
C.21 Atlantic cod, the 2011 population: Filtered FET regions 141

List of Tables

5.1 Python vs. C: CSS CMDS 53
5.2 Python vs. C: CSS SMACOF 54
5.3 Python vs. C: FET 55
5.4 Memory use, serial program: CSS 57
5.5 Memory use, parallel program: CSS 57
5.6 Memory use, serial program: FET 58
5.7 Memory use, parallel program: FET 58

C.1 Variable task size: CSS 117
C.2 Variable task size: FET 117
C.3 Variable number of threads: CSS 118
C.4 Scaled speedup: CSS 118
C.5 Variable number of threads, ts 25: CSS 118
C.6 Scaled speedup, ts 25: CSS 119
C.7 Variable number of threads: FET 119
C.8 Scaled speedup: FET 119

Abbrevations

Δ	Dissimilarity matrix, used for multi-dimensional scaling
σ_r	Raw stress
L_{10}FET	Negative base-ten logarithm of the Fisher's exact test score
BMI	Biomedicial Informatics research group at the Department of Informatics at the University of Oslo
chr	chromosome
CMDS	Classical multi-dimensional scaling
CPU	Central processing unit
CSS	Cluster separation score
DNA	Deoxyribonucleic acid
EDA	Ectodysplasin
FDR	False discovery rate
FET	Fisher's exact test
kb	kilo base (pairs)
MDS	Multi-dimensional scaling
mRNA	Messenger RNA
PRNG	Pseudo-random number generator
RAM	Random-access memory
RNA	Ribonucleic acid
SMACOF	Scaling by majorizing a complicated function
SNP	Single-nucleotide polymorphism
SOM/HMM	A self-organized map-based iterative Hidden Markov Model
VCF	Variant call format

Chapter 1

Introduction

1.1 Motivation

Comparative genomics is useful for finding the evolutionary relationships between different organisms. We can identify regions of parallel evolution, and gain a better understanding of evolution and the development of different species.

One way to identify the regions of parallel evolution is to compare the genome (genes) of individuals in two different populations. The goal is to find regions on the genome that are similar in all individuals in one population, but different in individuals of another population. These regions may possibly be linked to phenotypic traits in the two populations.

We have looked at two studies, one comparing marine-freshwater divergence in three-spined sticklebacks (Jones et al., 2012) and another comparing evolved *Drosophila* populations against their control population (Burke et al., 2010). Torkil Vederhus wrote a master thesis where he implemented tools for identifying divergent regions in the genome, and he compared his results with the results in the two articles (Vederhus, 2013). The tools were implemented in Python using the Genomic HyperBrowser framework. The tools, however, had some shortcomings, such as long computation time and usability of the tools, e.g. that the tool had to be run for each chromosome separately.

1.2 Research questions

The aim of this project is to make efficient tools that rapidly identify those regions in a genome of which the inter population variation show a significant correlation with their phenotypic traits. This is going to be done by improving the tools made by Vederhus (2013).

We are going to look at the following research questions:

- How much is it possible to speed up the computation by using different algorithms, programming languages and/or parallelization?

- How easy is it to integrate (parallel) C code into a large scale Python system?
- Is it possible to reproduce the results found by Vederhus, Jones et al. and Burke et al., and achieve good results on other data sets as well?
- How can a better user experience for the tools be made?

1.3 Overview of thesis

In chapter 2 a general background to this thesis is given, with a short introduction to the necessary biology, summaries of the stickleback study (Jones et al., 2012), the *Drosophila* study (Burke et al., 2010) and the thesis of Vederhus (2013), together with a short description of the Genomic HyperBrowser system. In chapter 3 the relevant methods used by Vederhus, Jones et al. and Burke et al. are described together with different methods and issues for parallelization. In chapter 4 the implementation of the tools are described, and in chapter 5 the results are presented; the new user interface, the speedup of the calculations and the results from the analyses. Chapter 6 contains the discussion of the results and the conclusion, while chapter 7 discusses possible applications for future work. Additional results from the analyses, together with an example run of the tools and a link to the source code, are included in the Appendices.

Chapter 2

Background

2.1 Essential biological concepts

The phenotype of an individual is its physical characteristics or traits, such as height or hair color (Klug, Cummings & Spencer, 2007). Our traits are carried from parents to child by genes. A mutation is any inheritable change and is a source of genetic variation. The alternative forms of a gene are called alleles. Different alleles can give different phenotypic traits (Klug et al., 2007).

The DNA carries genetic information. The DNA is a double helix, and each strand of the helix contains nucleotides. There are four different types of nucleotides, adenine (A), cytosine (C), guanine (G) and thymine (T). A and T and C and G are complimentary nucleotides. This means that the two strands of the double helix are complimentary.

Each gene is a recipe for a protein, and the protein has an important role in contributing to the phenotype. Different alleles can result in proteins that are slightly different and therefore result in different phenotypes. The process that creates proteins from genes is called the central dogma in genetics (Crick, 1970). The flow of information goes like this [1]:

$$\text{DNA} \xrightarrow{\textit{transcription}} \text{RNA} \xrightarrow{\textit{translation}} \text{protein}$$

The two main processes are:

1. **Transcription:** A part of the nucleotide sequence in one of the strands of DNA is transcribed to a complimentary strand of RNA, called messenger RNA (mRNA).

2. **Translation:** The information stored in the mRNA is used to produce a protein. A protein is made up of amino acids. The nucleotides in the mRNA form sequences of codons. Each codon consists of three nucleotides,

[1] Crick (1958) stated that *"Once 'information' has passed into protein it cannot get out again"* (p. 153), so this is a simplified description of the central dogma.

and codes for an amino acid. The RNA codes for 20 different amino acids (Alberts et al., 2014a).

Each codon has a corresponding transfer RNA (tRNA) which contains an amino acid. In this way the amino acids are linked in a long chain that creates a protein. There are $4^3 = 64$ different codons, so some codons code for the same amino acid. There are also codons that represents the start and the end of the gene (and therefore the protein).

Single-Nucleotide Polymorphism (SNP)

In this thesis we are going to look at a special type of variation in a gene, a single-nucleotide polymorphism (SNP). This is the most common type of genetic variation and is a single base change in a DNA sequence. For the base change to be considered as a SNP it must occur in a significant part of the population, for instance more than 1%. An example of a SNP is a position in the genome where some individuals in the population have a G-C nucleotide pair, while the rest has an A-T nucleotide pair (Alberts et al., 2014b).

These variations can occur in all parts of the genome, both coding (parts that codes for a protein) and non-coding regions (regions with less known function, probably regulatory function). In the human genome, most of the SNPs occur in non-coding regions, and only a few occur in coding regions (Alberts et al., 2014b). A SNP in a coding region changes the codon in this part of the DNA. There are more codons than amino acids, so some changes in the codon can make it code for the same amino acid, therefore having no visible effect. SNPs like this are called synonymous SNPs. Some SNPs, called non-synonymous SNPs, make the codon code for different amino acids. This leads to different proteins, which again can lead to phenotypic changes.

In this thesis we are going to make tools that identifies SNPs that differs between two populations. We are going to locate the relevant regions in the genome, and find SNPs that correlates with the different phenotypic traits in the two populations.

2.2 The genomic basis of adaptive evolution in threespine sticklebacks

The three-spined stickleback is a fish found in the Northern Hemisphere. The fish mostly lived in marine areas, but after the last ice age, when the ice drew back, it has evolved to live in freshwater streams and lakes. The fish adopted to freshwater and evolved changes in body shape, skeletal armour and many other phenotypic traits. Similar phenotypic traits have evolved in similar environments. This gives an indication that these traits have evolved by natural selection (Jones et al., 2012). The genetic basis for individual traits are already discovered, for instance the genes underlying the skeletal armour.

Jones et al. first obtained a reference genome for the three-spined stickleback. The reference genome was assembled from a female freshwater stickleback from

Bear Paw Lake, Alaska. They sequenced 20 additional individuals, from both freshwater and marine populations. These individuals were selected by looking at several individuals from different geographic locations. On these individuals Jones et al. used morphometric analysis to select individuals with typical marine and freshwater morphology (form and structure of the fish). Data from all the fish were pooled to identify SNPs. Jones et al. only used positions where at least four reads (four individuals) supported a variant allele. The goal was to find regions in the genome where freshwater fish were similar to each other, but different from marine fish. Jones et al. wanted to find variants in the genome that give evidence for repeated evolution across the two populations. One such loci has already been identified, variants of the ectodysplasin (EDA) locus which controls repeated armour evolution in freshwater populations (Colosimo et al., 2005), and they wanted to confirm that they could find that as well.

Jones et al. used two different methods:

- A self-organized map-based iterative Hidden Markov Model (SOM/HMM) to find the 20 most common patterns of genetic relationships between the 21 individuals.
- A genetic distance-based approach, where they built 21×21 pairwise nucleotide divergence matrices for windows over the genome. A window size of 2500 base pairs and a step size of 500 base pairs were used. The divergence between the populations was calculated with a cluster separation score (CSS) that quantifies the average distance between marine and freshwater clusters.

Both methods found the EDA locus that was connected with armour evolution in the top-scoring divergent regions. Other regions in the genome associated with marine-freshwater divergence were also found.

Jones et al. wanted to estimate the relative contributions of coding and regulatory changes in the genome. To do this they looked at 64 divergent regions, found with the use of both methods, and found that 41% mapped to non-coding regions (probably regulatory) of the genome, 17% to coding regions and 45% to both coding and non-coding regions.

The results suggest that parallel evolution of marine and freshwater stickleback occurs by dynamic reassembly of many islands of divergence, distributed across many chromosomes.

2.3 Genome-wide analysis of a long term evolution experiment with *Drosophila*

The common fruit fly (*Drosophila melanogaster*) has been used widely for biological research and is a common example of a model organism. A model organism is a non-human organism that is studied extensively, to understand biological phenomena. The knowledge is then transferred to other organisms.

Burke et al. re-sequenced populations of *Drosophila* that had been selected for accelerated development for 600 generations in a laboratory (Burke et al., 2010). The population developed 20% faster from egg to adult than the control population. They replicated this experiment for five separate populations and obtained sequences from three genomic DNA libraries: A pooled sample of the five replicated accelerated populations (ACO), a pooled sample of their control populaton (CO) and a single ACO replicate population (ACO_1). The goal was to identify SNPs that significantly differed between the ACO and CO populations. They aligned the reads to the reference genome of *Drosophila*, and only considered the regions with two different alleles. Burke et al. calculated a Fisher's exact test score for each SNP and ended up with 662 potential candidates for encoding differences between ACO and CO.

Burke et al. also did a genome-wide sliding window analysis, with widows of size 100-kb (kilo bases) and a step size of 2 kb. This was done to identify regions with divergence in allele frequencies between the ACO and CO populations and the ACO and ACO_1 populations. Significant divergence between the ACO and CO populations was found, but wery little evidence of divergence between the ACO and the single ACO_1 population was found.

2.4 The Genomic HyperBrowser

The Genomic HyperBrowser is an open-source and open-ended system for comparative genomic analysis. It can do inferential investigations; two annotations or tracks are compared to find a deviation from a null-model behaviour (Sandve et al., 2010). It is written in Python, and has a step-wise web based user interface. The HyperBrowser is tightly integrated with the Galaxy framework (Goecks, Nekrutenko and Taylor, 2010; Blankenberg et al., 2001; Giardine et al., 2005), and the standard Galaxy tools are available for use (Sandve et al., 2013). The HyperBrowser also has a framework for unit and integration tests.

The genomic data are represented as mathematical objects, and the biological investigations are performed as statistical analyses.

2.4.1 HyperBrowser jobs and tracks

In HyperBrowser the genome is stored as tracks and each track is split into bins. The length of the bin can be decided by the user, and is a given number of base pairs (bp) or a whole chromosome. The HyperBrowser analysis is divided in two phases: a local analysis, that works on each bin separately, and a global analysis (Sandve et al., 2010).

A run is started from the Galaxy web interface (https://hyperbrowser.uio.no), where the user selects the data set to work on, the size of the bins and the relevant analysis (Lillesæther, 2011). The job starts in `GalaxyInterface`, where a `StatJob` object is created from the specifications given by the user. In `StatJob`, `Track` and `Statistic` objects are created.

The computations start in `StatJob.run`. There are two phases of the analysis; the local analysis in the method `_doLocalAnalysis`, and the global analysis in the method `_doGlobalAnalysis`. For us, the local analysis is the most interesting, since it is here our part of the calculations occur.

2.4.2 Statistics

Statistic is a HyperBrowser module that defines a statistical operation on tracks. The name of a module has the form `<description>` + `Stat` (Lillesæther, 2011). Each statistic has two possible classes, of the type `Splittable` and `Unsplittable`. The unsplittable class is the actual implementation class, where the statistical analysis is implemented. The splittable version of the statistic can be used to divide the problem into smaller problems. The splittable class creates children that are unsplittable versions of the same statistic, and collects the results from these children.

The HyperBrowser code for local analysis looks like this:

```
def _doLocalAnalysis(self, results, stats):
    for region in self._userBinSource:
        res, stat = self._getSingleResult(region)
        results[region] = res
        # ...
    return stats


def _getSingleResult(self, region):
    stat = self._statClass(region, self._track, self._track2, \
    *self._args, **self._kwArgs)
    try:
        res = stat.getResult()
    # ...
    return res, stat
```

For each bin in the genome a track object and a statistic of the right type are created, corresponding to our choice of analysis. The results are computed with a call to the method `stat.getResult()`. This method eventually calls the `_compute` method in the right statistic subclass.

2.5 A Tool for Genome-wide Analysis of Genomic Divergence

Vederhus (2013) implemented several tools for identifying divergent regions in the genome across two populations. The tools were implemented in Python, within the Genomic HyperBrowser framework. Vederhus looked at the two

studies from the previous sections, the study of marine-freshwater divergence in three-spined sticklebacks (Jones et al., 2012) and the study of *Drosophila* populations from long term evolution experiments (Burke et al., 2010). Vederhus chose two of the methods used in the articles for identifying regions of genomic divergence across populations, The cluster separation score (CSS) and Fisher's exact test (FET). Both methods used sliding windows in the computations. In his thesis, a data structure for a sliding window over the genome, together with two statistics, one for each method, was implemented in the Genomic HyperBrowser. Vederhus also created several graphical user interfaces in HyperBrowser, two for each method, and one for converting the data to the correct file format.

To represent the SNP data Vederhus made his own specific file format. He used a custom version of the "value point" GTrack file format (Gundersen et al., 2011), where he used the following fields (Vederhus, 2013):

- Sequence id (or chromosome)
- Start (base pair address)
- Value of SNP
- Individual id

Vederhus had to convert the data files from both the stickleback study and the *Drosophila* study to get his chosen format.

He ran both tools on the stickleback data and the *Drosophila* data, and compared his results with the results found by Jones et al. and Burke et al. Vederhus identified many of the same regions in the genome.

2.5.1 Tool implementation in HyperBrowser

Vederhus (2013) made several tools for his analyses, the most important being the CSS tool and the FET tool. The tools were run for each chromosome separately, needing two files per chromosome, one for each population group. As mentioned in section 2.4, the HyperBrowser divides the genome into bins, and Vederhus created one bin per chromosome. The methods for analysing genomic divergence were implemented in HyperBrowser using two statistics; `CategoryClusterSeparationStat` implemented the CSS method, while the `FisherExactScoreStat` implemented the FET method. In each statistic, a sliding window was created for the chromosome. The statistic looped through the windows (slided through the chromosome), and for each window, the CSS or FET score was calculated, and then the window was moved a given number of base pairs. The output of the tool was written to a file, and the results could be filtered to find relevant regions with other tools made by Vederhus.

2.5.2 Issues with the current tool

The main issues with Vederhus' implementation were the time it took for the tool to analyse the chromosome and the usability of his tools. At the time of Vederhus' writing, the HyperBrowser loaded all data into memory during preprocessing. The data file for the stickleback analysis was too big, requiring more than 50 GB of data loaded into the virtual memory (Vederhus, 2013). Therefore, Vederhus had to divide the SNP data into chromosomes, and run the analysis on each chromosome separately. This led to much manual work.

The implementation was done in Python, and even though NumPy was used to speed up the computation, the execution time was too slow. For the largest stickleback chromosome, chromosome I, the cluster separation score (CSS) tool took 45 hours to run (Vederhus, 2013). The implementation therefore has to be speeded up, with better algorithms and/or parallelization. The lack of computational speed was another reason for dividing the data files. With one file per chromosome, Vederhus could manually parallelize the analyses, and thus speed up the calculation.

Vederhus tested his tool with biology students, and found several issues with the user friendliness of his tools. Some of the issues lay in the help page of the HyperBrowser, which had too complex examples. Vederhus suggested naming the tools better and creating a simple FAQ for the tool on the tools' web site. The file format was not user-friendly for the biologists. It was cumbersome to divide the data files into chromosomes, and according to Vederhus, the file formats used in bioinformatics lack standardization, which makes the job of finding a good file format difficult.

2.6 Performance metrics

Here follows a short description of different performance metrics for parallel programs (Grama, Gupta, Karypis & Kumar, 2003):

2.6.1 Runtime

The *serial runtime* of a problem, T_S, is the total amount of time taken by the problem; the time elapsed between the beginning and the end of the program. The *parallel runtime* of a problem, T_P, is the total amount of time taken from the beginning of the parallel program, until the last thread (or process) is finished (Grama et al., 2003).

2.6.2 Overhead

The total time used by all the processes in a parallel system is

$$p * T_P$$

where p is the number of processes.

If T_S is the serial runtime of the fastest known serial algorithm of the problem, then the *total parallel overhead* is given by:

$$T_O = p * T_P - T_S$$

2.6.3 Speedup and efficiency

The *speedup* is given by

$$S_P = \frac{T_S}{T_P}$$

Linear speedup is given when $S_P = p$, and is, theoretically, the best speedup one can achieve. Then $T_P = \frac{T_S}{p}$.

The *efficiency* of the program is given by

$$E = \frac{S_P}{p}$$

In an ideal parallel system, speedup is equal to p and the efficiency is equal to 1, but in practice, the efficiency is between 0 and 1.

Chapter 3

Methods

In this section, the two methods used to analyse genomic divergence in Vederhus (2013), together with the relevant algorithms, are described. Parallel programming concepts that may be used to speed up these calculations are also presented.

3.1 Cluster Separation Score (CSS)

Jones et al. (2012) divided the genome into windows of size 2500 bp, with a step size of 500 bp. For each window, they built a 21×21 dissimilarity (distance) matrix, Δ. Each δ_{ij} represented a pairwise nucleotide divergence (π) between two fish. Jones et al. used all positions in the genome that passed the *4-read criteria* (a validated position). If the δ_{ij} had less than 100 validated positions, the comparisons were discarded. These positions were instead filled with the average value of the distance matrix. For a given window, if more than 105 of 210 comparisons were discarded, the entire window was excluded from analysis (Jones et al., 2012).

For each pairwise nucleotide divergence matrix Δ, Multi-dimensional scaling (MDS) was performed to extract the two major axis of genetic divergence. The goal was to find two separate clusters, one for marine and one for freshwater fish. To measure the strength of the divergence between the two clusters in the two-dimensional MDS space, Jones et al. used a cluster separation score (CSS) between the marine and the freshwater population. The CSS measure is given by (Jones et al., 2012):

$$CSS = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} s_{i,j}}{mn} - (m+n)(\frac{\sum_{i=1}^{m-1} s_{i,i+1}}{m^2(m-1)} + \frac{\sum_{j=1}^{n-1} s_{j,j+1}}{n^2(n-1)})$$

where m and n are the size of the marine and freshwater population, i and j are individuals from the respective populations and s is the Euclidean distance between a pair of fish in the first two axes of the MDS space. The first term

corresponds to the average between-group distance, the second to the within-marine group distance, and the third to the within-freshwater group distance (Jones et al., 2012). A large CSS score indicates strong and parallel divergence between the two groups, while a score near 0 indicates no parallel divergence between the groups. A negative score is possible if the two groups are more similar to each other that they are to individuals within their own group (Jones et al., 2012).

To measure the significance of the CSS, the *p*-value was calculated. The relevant regions were filtered with a *p* cutoff threshold based on a false discovery rate (FDR) of 0.02 or 0.05 (Jones et al., 2012).

There are two main problems with this approach:

- Calculating MDS is relatively expensive. Classical MDS has a complexity of $O(N^3)$ (Marsland, 2009), and involves matrix multiplication and computation of eigenvectors and eigenvalues.
- Calculating the *p*-value is also expensive. According to Vederhus (2013), this is the most resource-intensive calculation of all. To calculate the p-value all the CSS values for all the different divisions of the set have to be calculated. This is not feasible. The amount of possible divisions are defined as:

$$\frac{n!}{r!(n-r)!}$$

  where n is the total amount of individuals and r is the size of one group (Vederhus, 2013). Vederhus tried to solve this problem by using a Monte Carlo method to estimate the *p*-value.

The CSS method consists of three steps (Vederhus, 2013). First, one has to measure the distance of each individual and store it in the dissimilarity matrix Δ. This is done by using a suitable distance metric. Then, the distance data are scaled to two dimensions, using multi-dimensional scaling. The two dimensional data are then scored by a cluster separation score.

3.1.1 Multi-Dimensional Scaling (MDS)

The goal of multi-dimensional scaling is to represent similarity or dissimilarity in the data as distances between points in a lower dimensional space (Borg & Groenen, 2005), usually two or three dimensions. It can for instance be used to better visualize the data, or to find structures in the data set. There are two main types of models, metric and non-metric MDS. In metric MDS the actual distances/ratio between the distances are used, but in non-metric MDS only the rank order of the points are taken into consideration. Vederhus (2013) found better results with metric MDS (less noise and stronger signals in known areas of genomic divergence (Vederhus, 2013)), thus non-metric MDS will not be considered here.

Two different methods for multi-dimensional scaling are classical MDS and iterative MDS. Classical MDS gives an easy algebraic solution and is easy to implement. Classical MDS assumes that the distances between points are Euclidean and generates a best-possible solution based on a minimizing criterion named Strain. Unfortunately, it does not work well on all data sets, and has a complexity of $O(N^3)$ (Marsland, 2009). A more optimal solution would be to use an iterative algorithm. The currently best of these are the SMACOF algorithm, where a criterion named Stress (Borg, Groenen & Mair, 2013) is minimized. Iterative MDS is more flexible than classical MDS, but iterative methods are not guaranteed to find the global optimum, since they often get stuck in local optima (Borg et al., 2013).

Vederhus used the Python function `sklearn.manifold.MDS` to calculate MDS, which uses the SMACOF algorithm. Jones et al. (2012) did not describe which method they used. We have tried to contact them, but so far we have not received any response.

In the following two sections, three different methods for multi-dimensional scaling are described; classical MDS, iterative MDS and a combination of the two.

3.1.2 Classical MDS

Classical multi-dimensional scaling (CMDS), also known as Torgerson scaling, works well when the dissimilarity data are Euclidean distances, or when you can assume that the data are Euclidean distances (Borg et al., 2013).

The goal of MDS is to extract the two major axis of genetic divergence in the data. The data are represented by a dissimilarity matrix Δ. For the stickleback data, this is a matrix of count differences. The number of dimensions to scale down to is given by m. The algorithm consists of the following steps:

1. Square the dissimilarity matrix. This is simply Δ^2.

2. Convert the dissimilarity matrix to scalar products. This is done through double centering of Δ^2. The centering matrix $\mathbf{Z}$ is given by:

   $$\mathbf{Z} = \mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^T$$

   where $\mathbf{I}$ is the identity matrix and $\mathbf{1}$ is a column vector with n ones. n is the dimension of the dissimilarity matrix. The matrix of scalar products is then given by

   $$\mathbf{B}_\Delta = -\frac{1}{2}\mathbf{Z}\Delta^2\mathbf{Z}$$

3. Get the eigen decomposition of the matrix $\mathbf{B}_\Delta$:

   $$\mathbf{B}_\mathbf{D} = \mathbf{Q}\Lambda\mathbf{Q}^{-1}$$

Λ is a diagonal matrix with the eigenvalues on the diagonal, and the columns of $\mathbf{Q}$ is the corresponding eigenvectors.

4. To scale down to m dimensions, the first m eigenvalues greater than 0 and the corresponding eigenvectors are used. This is stored in Λ_+ and $\mathbf{Q}_+$. The solution $\mathbf{X}$ is then given by

$$\mathbf{X} = \mathbf{Q}_+ \Lambda_+^{1/2}$$

3.1.3 Iterative MDS

The SMACOF algorithm (Scaling by MAjorizing a COmplicated Function), is currently the best algorithm for iterative MDS (Borg et al., 2013). The goal of MDS is to find a lower-dimensional configuration of points representing the objects such that the distance between any two points matches their dissimilarity as close as possible. With the SMACOF algorithm this is done by miminzing a Stress function for all possible MDS representations $\mathbf{X}$ (Borg & Groenen, 2005).

The dissimilarity of two objects i and j is given by δ_{ij} and indiciates how dissimilar the two objects are. A large δ_{ij} indicates that i and j are very dissimilar, while a small value indicates similarity.

The Euclidean distance between two points in $\mathbf{X}$ is given by (Borg & Groenen, 2005):

$$d_{ij} = [\sum_{a=1}^{m} (x_{ia} - x_{ja})^2]^{1/2}$$

Raw Stress, σ_r, first defined by Kruskal (1964) gives a measure of the total error of the MDS representation $\mathbf{X}$. It is given by (Borg & Groenen, 2005):

$$\sigma_r(\mathbf{X}) = \sum_{i<j} w_{ij}(d_{ij}(\mathbf{X}) - \delta_{ij})^2$$

Since both the Euclidean distances and dissimilarities are symmetric, it is enough to look at half of the values. In some applications, the dissimilarity δ_{ij} is not defined for all i, j. Therefore the weight w_{ij} is introduced. It is 1 for all defined values of δ_{ij}, and 0 for undefined values (Borg & Groenen, 2005). In this application, all the dissimilarity values are defined, so $w_{ij} = 1$.

To find the best possible representation $\mathbf{X}$, $\sigma_r(\mathbf{X})$ is minimized over $\mathbf{X}$. The minimization is done by a method called iterative majorization, based on the work of De Leeuw (1977). This method generates monotonically nonincreasing function values. If the function is bounded below, one will end up in the local minimum (Borg & Groenen, 2005). Unfortunately, there is nothing that indicates how fast the method will converge to a minimum. Therefore, an algorithm based on iterative majorization is not necessarily very fast (Borg & Groenen, 2005), perhaps not faster than classical MDS.

Scaling by MAjorizing a COmplicated Function (SMACOF)

In the following we assume that $w_{ij} = 1$ for all i, j. The SMACOF algorithm consists of the following steps (Borg & Groenen, 2005):

1. Set $\mathbf{Z} = \mathbf{X}^0$, where $\mathbf{X}^0$ is a (non)random start value. $k = 0$ and ϵ is a small positive constant, for instance $\epsilon = 1e^{-6}$.

2. Compute the Stress of the start configuration:

$$\sigma_r^0 = \sigma_r(\mathbf{X}^0)$$

   Set $\sigma_r^{-1} = \sigma_r^0$.

3. While $k = 0$ or $(\sigma_r^{k-1} - \sigma_r^k > \epsilon$ and $k \leq$ maximum iterations) do

   4. Increase k with one.
   5. Compute the Guttman Transform $\mathbf{X}^k$:

$$\mathbf{X}^u = \frac{1}{n}\mathbf{B}(\mathbf{Z})\mathbf{Z}$$

   where the elements of $\mathbf{B}(\mathbf{Z})$ are given by

$$b_{ij} = \begin{cases} -\dfrac{w_{ij}\delta_{ij}}{d_{ij}(\mathbf{Z})}, & \text{for } i \neq j \text{ and } d_{ij}(\mathbf{Z}) \neq 0 \\ 0, & \text{for } i \neq j \text{ and } d_{ij}(\mathbf{Z}) = 0 \end{cases}$$

   and

$$b_{ii} = -\sum_{j=1, j\neq i}^{n} b_{ij}$$

   6. Compute $\sigma_r^k = \sigma_r(\mathbf{X}^k)$.
   7. Set $\mathbf{Z} = \mathbf{X}^k$.

The SMACOF algorithm has several nice properties. It can be showed that $\mathbf{X}^k$ converges linearly to a stationary point (De Leeuw, 1988). It handles zero distances well, and if there are no zero distances, the Guttman Transform is a steepest descent step with a fixed step size (Borg & Groenen, 2005).

A common problem with iterative methods like SMACOF is that they tend to only reach local optima. The start point controls the search area of the algorithm, and therefore which optimum that is reached. A possible way to increase the chance of reaching a global optimum is to run the algorithm with several different random start points. This is the solution chosen here.

Combining SMACOF with CMDS

Another possible way to avoid local optima is to use a better starting configuration. A possible solution is to use the result from the classical MDS as the starting point of the algorithm (Borg et al., 2013).

3.1.4 Statistical significance

To measure the significance of the results, we have to calculate the p-value. When a statistical analysis is performed, we have to define a null hypothesis, and an alternative hypothesis. The goal is to be able to discard the null hypothesis. The p-value is the probability of getting the observed results, or something more extreme, given that the null hypothesis is true. If this probability is low enough, the results are statistically significant, and one can discard the null hypothesis. Often,

$$p < 0.05$$

is considered significant, while a p-value greater or equal to 0.05 is not significant.

Monte Carlo Tests

Jones et al. (2012) calculated the statistical significance by calculating the CSS for each possible grouping of individuals. As Vederhus (2013) pointed out, this is not feasible. Instead, Vederhus used a Monte Carlo test to estimate the significance. The following description of Monte Carlo tests are taken from Phipson and Smyth (2010).

In Monte Carlo tests, independent data sets (permutations of the original data set) are created under the null hypothesis by simulation using pseudo-random numbers. If the observed test statistic is given as t_{obs} and the simulated test statistic is t_{sim}, the ideal p-value is given by:

$$p_\infty = P(t_{sim} \geq t_{obs})$$

The creation of an infinite number of simulated data sets is not possible, therefore n data sets are generated. Phipson and Smyth (2010) showed that the exact Monte Carlo value is given by:

$$p = \frac{r+1}{n+1}$$

where r is the number of simulated test statistics $t_{sim} \geq t_{obs}$.

When the sampling is done with replacement, as is the case in this application, each permutation of the data set is generated independently, making it possible for the permutations to repeat themselves. In this case, the exact p-value is slightly less than $(r+1)/(n+1)$, since the original data and test statistic t_{obs} could be included at least once in the permutations. Therefore, as showed by Phipson and Smyth (2010), the Monte Carlo test tend to over-estimate the

p-value, creating too conservative values. When n increases, the exact p-value converges to $(r+1)/(n+1)$, therefore it is important to use a large n.

3.2 Fisher's Exact Test (FET)

Fisher's exact test (FET) is a statistical significance test. It is used on two nominal (categorical) variables, X and Y (McDonald, 2014). In this application, X could be the type of fish (saltwater or freshwater), and Y could be the major or minor allele (Vederhus, 2013). FET measures the significance of the observed data. The FET is calculated on a 2×2 table of numbers, with a null hypothesis that the relative proportion of a variable in the table is independent of the other, for instance that the number of major alleles is independent of the type of fish.

An example table:

	Freshwater	Saltwater	
Major	a	b	$R_1 = a + b$
Minor	c	d	$R_2 = c + d$
	$C_1 = a + c$	$C_2 = b + d$	$N = a + b + c + d$

The conditional probability of getting this particular matrix is given by the hypergeometric distribution (Feldman & Klinger, 1963):

$$P_{cutoff} = \frac{(a+b)!\,(c+d)!\,(a+c)!\,(b+d)!}{N!\,a!\,b!\,c!\,d!}$$

To find the p-value, the conditional probability for all matrices is calculated, where the sums of the rows and columns (R_1, R_2, C_1, C_2) are consistent. For a 2×2 table the p-value can be found by taking the sum of all the matrices with $p \leq P_{cutoff}$. If $p < 0.05$, the result is statistically significant, and we can conclude that there is a connection between alleles and fish type.

Burke et al. (2010) calculated the FET for each SNP position in the genome, and took the negative base-ten logarithm of the p-value ($\mathrm{L_{10}FET}$). They used windows of size 100 kb with a step size of 2 kb, and calculated a $\mathrm{L_{10}FET_{5\%Q}}$ score for each window: the 95th percentile of the scores in the window. The standard deviation of 100 bootstrap replicate samples of $\mathrm{L_{10}FET_{5\%Q}}$ was also calculated. Burke et al. (2010) selected the top scoring regions with the following formula:

$$\mathrm{median}(\mathrm{L_{10}FET_{5\%Q}}) + \mathrm{qnorm}(0.999) \times \mathrm{quantile}(\sigma, \mathrm{probs} = 0.75)$$

where σ is the list of standard devations over all windows, the median is taken over the $\mathrm{L_{10}FET_{5\%Q}}$ scores for all windows and qnorm(0.999) is the 99.9th percentile of the standard normal distribution. The problem with the FET is that it is expensive to calculate the hypergeometric distribution for large values, since the computation of the factorial is time consuming (Zar, 1987).

Vederhus did not compute the complete FET, only the value P_{cutoff}. The standard deviation calculation was not complete, either; Vederhus calculated

the standard deviation for all the scores in the window, instead of the standard deviation of 100 bootstrap replicate samples of $L_{10}FET_{5\%Q}$.

3.2.1 Fisher's Exact Test algorithm

The Fisher's exact test (FET) is calculated from a 2×2 contingency table:

a	b	$R1 = a + b$
c	d	$R2 = c + d$
$C1 = a + c$	$C2 = b + d$	n

where $n = a + b + c + d$.

The probability of observing the values by chance, given that rows are independent from columns, is:

$$P_{cutoff} = \frac{(a+b)!\,(c+d)!\,(a+c)!\,(b+d)!}{N!\,a!\,b!\,c!\,d!}$$

which can be written as

$$P = \frac{\binom{a+b}{a}\binom{c+d}{c}}{\binom{n}{a+c}}$$

where

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

is the binomial coefficient. Calculation of the factorial is time consuming, but we can efficiently calculate the binomial coefficient by the formula (Zar, 1987):

$$\binom{n}{k} = \frac{n(n-1)(n-2)\dots(n-k+1)}{k(k-1)(k-2)\dots1}$$

remembering that $\binom{n}{n} = \binom{n}{0} = 1$ and that $\binom{n}{k} = \binom{n}{n-k}$.

To get the complete two-tailed FET score, the probabilities of each more extreme table have to be calculated and added together. A short cut for calculating the two-tailed FET was proposed by Feldman and Klinger (1963) and explained more detailed by Zar (1987). This is described in the following paragraph.

A short cut calculation for FET Given a $s \times 2$ table of frequencies:

a_0	b_0	$a_0 + b_0$
c_0	d_0	$c_0 + d_0$
$a_0 + c_0$	$b_0 + d_0$	N

where $a_0 \leq b_0$, c_0, d_0. The two tailed probability is calculated by calculating the FET for the two tails, and adding the probailities together.

To calculate the one-tailed Fisher's exact test, probabilities for all $a_0 + 1$ tables where the value in their principal diagonal is a_0, $a_0 - 1$, ..., 0 have to be calculated (Feldman & Klinger, 1963). Given that $a_i = a_0 - i$, we only have to calculate the score of the original table with the general formula for P_{cutoff}:

$$P_0 = \frac{\binom{a_0+b_0}{a_0}\binom{c_0+d_0}{c_0}}{\binom{N}{a_0+c_0}}$$

the probabilities for the rest of the tables can be calculated by

$$P_{i+1} = \frac{a_i d_i}{b_{i+1} c_{i+1}} P_i$$

where

$$a_{i+1} = a_i - 1, \quad d_{i+1} = d_i - 1$$
$$b_{i+1} = b_i + 1, \quad c_{i+1} = c_i + 1$$

The summation of all these probabilities

$$P = \sum_{i=0}^{a_0} P_i$$

is the one-tailed probability for the test (Zar, 1987).

To calculate the two-tailed probability, the probability of the other tail has to be calculated. This is done with the following steps; First, find the most extreme table computed in the first tail. This is the last table we computed, with $a_i = 0$. Then, find the smallest margin total of this table, m_1, together with the smallest marginal total of the opposite margin, m_2. The frequency associated with these two marginal totals is called f, and the value in this table cell is changed to $m_1 - f$. The rest of the values in the table are updated accordingly, to make sure the row and column totals stay the same.

The table is rotated, so that a_0 is the smallest frequency, and b_0, d_0 and c_0 follow in clockwise order. This table is the most extreme table in the second tail. The probability of this table, Q_0, is calculated by the binomial formula:

$$Q_0 = \frac{\binom{a_0+b_0}{a_0}\binom{c_0+d_0}{c_0}}{\binom{N}{a_0+c_0}}$$

If $Q_0 > P_0$, the probability of the original table, the calculation is stopped and the two-tailed probability is equal to the one-tailed probability:

$$P_{two\text{-}tailed} = P$$

If that is not the case, the probability of the next table is computed by the formula (Zar, 1987):

$$Q_{i+1} = \frac{b_i c_i}{a_{i+1} d_{i+1}} Q_i$$

where

$$a_{i+1} = a_i + 1, \quad d_{i+1} = d_i + 1$$
$$b_{i+1} = b_i - 1, \quad c_{i+1} = c_i - 1$$

The calculation of new probabilities continues until the new probability, Q_{i+1}, is larger than the probability of the original table, P_0.

We sum all the probabilities together,

$$Q = \sum Q_i$$

and add the result to the one-tailed probability P to get the total two-tailed Fisher's exact test:

$$P_{two\text{-}tailed} = P + Q$$

There are some additional short cuts that can be made. If the row or column totals are equal ($R1 = R2$ or $C1 = C2$), the two-tailed score is simply the double of the one-tailed score: $P_{two\text{-}tailed} = 2P$ (Zar, 1987).

3.3 Parallel programming systems

In this thesis parallelization is going to be used to increase the speed of the calculations. The following sections includes descriptions of possible ways to do this.

Parallel computing is the use of a parallel computer to speed up the computation of a single problem. There are two main types of parallel computers. A multicomputer is a parallel computer made out of many processors, each with its own memory. They communicate by sending messages to each other. A centralized multiprocessor is a more integrated system, where each CPU share a global memory space (Quinn, 2003). In the following sections these two types of parallelization will be described, to see how they can be used in this thesis.

3.3.1 Message Passing Interface (MPI)

The message-passing programming model is characterized by a partitioned address space (Grama et al., 2003). The hardware is (logically) divided into several processes, each with its own local memory. The processes can communicate by communication links. An interaction between two processes requires cooperation from both processes - the process providing the data (sending) and the process accessing the data (receiving). All the data must be partitioned, and this together with the need for communication give increased complexity to the parallel program.

The most popular message-passing library is MPI, the message-passing interface. This describes a standard library for message-passing that is implemented in several programming languages, including C.

Task/channel model

The message-passing paradigm can be represented by the task/channel model. In the task/channel model a parallel program is represented as a collection of tasks (processes) that can interact with each other through channels (communication links). A task is a program with a local memory. The memory contains the instructions of the program and its private data. A task can communicate with other tasks through I/O ports, it can send data through the output port and receive data through the input port. A channel connects the output port of one task with the input port of another, and provides communication between tasks (Quinn, 2003).

In the message-passing paradigm the address space is divided, so the data and the tasks need to be divided between the different processes. This requires some planning, to utilize the parallelization in the best possible way, and to minimize overhead. Overhead is caused by to two things that doesn't exist in a sequential algorithm: communication between processes and idle time spent by a process (Grama et al., 2003).

Ian Foster has described a methodology for designing parallel algorithms for the task/channel model. It consists of four steps: Partitioning, communication, agglomeration and mapping (Quinn, 2003).

Partitioning In this stage the computations are divided in individual tasks. The goal is to get as many individual tasks as possible, and we need to split both the computation and the data into small pieces. There are two main strategies here, domain decomposition and functional decomposition. In domain decomposition the data are first divided into pieces, and then, this data partitioning is used to find a partitioning of the computations (Grama et al., 2003). This is often used for matrix-decomposition (for instance matrix multiplication). With functional decomposition the computations are divided before the data. This often leads to concurrency through pipelining.

Communication When the tasks are partitioned, the necessary communication between them needs to be determined. Local communication is confined to a few processes communicating with each other, but with global communication many processes are involved in the communication. The communication is a part of the overhead of the parallel program, and needs to be minimized.

Agglomeration In this step several tasks are grouped together into larger tasks. This is done with several goals in mind:

- To minimize communication overhead. This can be done by increasing the locality of the communication, or reducing the number of messages being sent. It is better to send larger and fewer messages, because each communication has a startup cost, the message latency.

- To maintain scalability of the problem.
- To reduce engineering costs.

Mapping Finally, the tasks can be mapped onto the different processors. The goals are to maximize processor utilization (by balancing the load evenly) and minimize communication overhead. These are often conflicting goals. There are several common mapping strategies, each suitable to be used on a certain type of problem.

3.3.2 Pthreads

Pthreads is a standardized model for writing concurrent programs with threads in a shared-memory environment. The underlying hardware of a shared-memory environment consists of many processors, each with access to the same shared memory. Pthreads is an easy, portable way to specify concurrent applications, and is built on the UNIX C programming interface (Nichols, Buttlar & Farrell, 1996). The P stands for POSIX, Portable Operating System Interface.

Why use threads? Because threads are cheaper to create and use than processes. Since threads share a global memory environment, little or no communication is needed. Thus, the creation of threads, communication and synchronization of shared data creates less overhead with threads than with processes. Pthreads is a standard for concurrent programming, which differs from parallell programming, where the tasks are run at the same time, in true parallel. With Pthreads the tasks that can be run concurrently are defined; whetether they are run concurrently or parallel depends on the operating system and hardware the threads are run on.

A threaded program starts in a single thread, called the main thread. This thread (and all other threads) can create new threads, with a call to the method `pthreads_create`:

```
pthread_t thread;

pthread_create(&thread,
        &thread_attribute_object,
        (void *) method_to_be_executed,
        (void *) &thread_argument);
```

This method takes four arguments:

- A pointer that identifies the newly created thread, of type `pthread_t`.
- A thread attribute object. This specifies different characteristics for the thread. `NULL` here gives the default setup.
- A pointer to the method to be executed by the thread. This method is where the thread starts its execution, and the thread exits when the

method is finished (much like how the `main` method functions for the main thread).

- A pointer to a parameter sent to the method. Threads can only start in methods that have only one parameter, therefore this parameter is often a `struct` with the necessary information.

All the threads created (with a small exception of the main thread) are equal, and there are no special relationships between threads. Since the threads operate in a shared-memory environment, one must be able to synchronize their execution. This can be done with `pthread_join`. This method can be called by any thread, but it is often called by the main thread, waiting for the other threads to finish so it can do some cleanup.

In a shared-memory environment the threads can have shared variables that they need to read and write to. To make the variable thread safe, and avoid concurrent updates, mutex variables are used. Mutex stands for mutually exclusive, and let threads mutually exlude one another from certain variables.

Before a thread writes to a shared variable, the mutex lock is aquired. Then, the shared variable is written, before the thread releases the lock. Other threads trying to aquire the lock, must wait until the lock is released. There are no direct lock on the shared variable itself, so it is up to the programmer to make sure that all writes to a shared variable is controlled by a mutex. An example of this (Nichols et al., 1996):

```
pthread_mutex_t var_mutex = // ..
int shared_var = 0;   // global shared variable

void method(...) {
    pthread_mutex_lock(&var_mutex);
    // code for writing to shared_var
    pthread_mutex_unlock(&var_mutex);
}
```

A thread can terminate when

- The executed method is finished and terminates
- The thread calls `pthread_exit`
- Another thread calls the method `pthreads_cancel` on this thread

By monitoring the different exit values from each thread, one can see if a thread exited normally or not. This is important for proper error handling.

3.3.3 OpenMP

The OpenMP interface is used in a shared-memory environment. OpenMP is used with compiler directives, and works great for fork/join parallelism: At the beginning of the execution, a single thread execute code sequentially. This thread is called the master thread. When the program needs parallelization, the master thread forks (creates or awakens) new threads. The new threads and the master thread works concurrently throughout the parallel section. Afterwards, the threads are joined (killed or suspended), and only the single master thread continues executing the code.

In the message-passing model the number of processes are often static during the execution of the program, but with the shared-memory model the number of active threads varies dynamically. One of the advantages over MPI is that OpenMP supports incremental parallelism. Each block of code can easily be parallelized, independently of the rest of the code. This makes it easy to gradually parallelize the program, which makes OpenMP ideal for agile development methods, such as Scrum (Sommerville, 2011). With MPI the entire problem has to be parallelized at once - and this takes a lot of planning. The four steps in Foster's methodology are good examples of that.

Shared memory enviroments creates other challenges than with the message-passage paradigm, such as race conditions and non-deterministic behavior.

Combining MPI and OpenMP/Pthreads

Many multicomputers are actually collections of centralized multiprocessors (Quinn, 2003). In these cases it can be a good idea to combine MPI with OpenMP or Pthreads. Each multiprocessor can have its own MPI process, while the program inside the multiprocessor can be run using threads and a shared-memory model. This can often give faster programs than pure MPI-programs, because there is lower communication overhead.

3.3.4 MapReduce

The programming model

MapReduce, introduced by Dean and Ghemawat (2008), is a programming model that simplifies the processing of large data sets. In processing of large data sets, the computations have to be distributed over several machines. Parallelization, distribution of data, machine failures and much more have to be taken into account, which can make the code extremely complicated. MapReduce solves this by abstracting away the complex distribution of data, and offers automatic parallelization and execution of programs on large machine clusters (Dean & Ghemawat, 2008).

The user specifies two functions, *map* and *reduce*. *Map* takes a key/value pair as input and generates a set of intermediate key/value pairs. The *reduce* function takes an intermediate key and a list of corresponding values, and merges the values to a smaller set, typically of size one or zero. The MapReduce library

takes care of the collection of intermediate key/value pairs produced by the *map* function. It groups the values corresponding to the same key into a set, which it then passes on to the *reduce* function. A typical example of the map/reduce functions is shown below (Dean & Ghemawat, 2008):

```
map (k1, v1)            -> list(k2, v2)
reduce (k2, list(v2))   -> list(v2)
```

The advantages of MapReduce is that it is easy to use. Programmers without any experience of parallel or distributed computing can use it without any problems. Several common problems can be written in the map/reduce format, which makes it useful for a broad range of applications. The disadvantage with MapReduce is that it is only usable for problems that can be (easily) written as maps and reduces.

3.4 Parallel design models

3.4.1 The boss/worker model

In the boss/worker model, one thread acts as the boss; it continually takes in inputs, and delegates tasks to worker threads. This can be done in two ways. The boss can either create new threads dynamically with each incoming task, or create all threads when the program starts. This is called a thread pool. When a new task comes in, the boss notifies the workers (put the task on the queue), and the workers pick available tasks from the queue. This saves some overhead with thread creation during execution. This model works well for servers. To minimize overhead, it is important to minimize the communication between the boss and the workers, and between two and two workers (Nichols et al., 1996).

3.4.2 The peer model

In the peer model, all threads work concurrently on the tasks without a specific leader (Nichols et al., 1996). One thread, the 'main' thread, creates the other threads, and works together with them on the tasks, or waits until the other threads are finished. The input is often static, and each thread is responsible for its own input, for instance a part of an array. The peer model is suitable for applications with fixed/well defined input.

3.4.3 The pipeline model

The pipeline model works for typical assembly types of applications, with a stream of inputs, a series of sub-operations (or filters), where each stage can handle a different unit of input at a time (Nichols et al., 1996). In this model the overall throughput of tasks are limited to the stage that takes the longest time, so it is important to try to balance the amount of work in each stage.

3.5 Mapping of tasks in parallel systems

3.5.1 Overhead

The goal of a good mapping of tasks is to minimize the overhead of a program. There are two main sources of overhead (Grama et al., 2003):

- Interaction between threads, such as communication and synchronisation.
- Idle time spent by a thread, by waiting for another thread to finish (due to synchronization), or by some threads finishing long before others.

The communication time and idle time spent by a thread need to be minimized, but these goals are often conflicting. A minimum amount of communication is achieved when one thread do all the computations, but this is bad load balancing, leaving many threads idle. A good mapping must be able to compromize between these two issues, to balance the load and minimize the overhead due to communication at the same time.

3.5.2 Tasks and work division

There are two main ways of mapping the tasks on to different processes:

- Static mapping
- Dynamic mapping

With static mapping, an equal amount of tasks are given to each process. This works well when each task is equally big, but for tasks of unequal computational size, a dynamic mapping is better. With dynamic mapping, the tasks are divided during the execution of the program.

Chapter 4

Implementation

4.1 Implementation choices

Vederhus (2013) wrote several HyperBrowser tools:

- **Fisher Exact Test SNP Tool**
- **Filter Fisher Scores**
- **Cluster Separation Score**
- **Significant CSS Regions**
- **Convert Stickleback Snps to Gtrack**

and two statistics, one for each method, FET and CSS:

- **CategoryClusterSeparationStat**
- **FisherExactScoreStat**

We have made small changes to all of these tools, but the largest modifications were done to the the **Fisher Exact SNP Tool**, the **Cluster Separation Score** and the corresponding statistics. The main computations were moved from Python to C, and the C code is parallelized in a shared memory environment.

A new tool was created, to convert VCF files (Danecek et al., 2011) to the custom GTrack file format used by our tools.

4.1.1 Main structure of the tools

The tool can now be run on a genome-wide basis. The input is two files per genome, one for each population group. Each HyperBrowser bin is the size of a chromosome, which means that the analysis for each chromosome is run

serially, one chromosome after the other. For each chromosome, a statistic is created. This statistic collects the necessary data before it calls the parallel C code, which is called from Python with the help of Cython. The parallel code was written with Pthreads. A serial version of the tool was also created, to be able to study how much speedup can be achieved in C compared to Python.

4.1.2 Methods

The method for calculating CSS presented by Jones et al. (2012) was implemented, together with all three methods described in section 3.1 for calculating MDS. This enabled us to study the differences between the three methods, both in speed and statistical power. We kept the distance metrics described by Vederhus (2013), where, for non-frequency data, only the positions with major and minor allele are counted. Positions where both reads are observed are discarded. This will give a more conservative statistic, with less noise in the results (Vederhus, 2013). A Monte Carlo test was used for estimating p-values, to get a faster calculation. For the Fisher's Exact Test, the fast algorithm described in section 3.2 was used, together with the formula of the limit used by Burke et al. (2010).

4.1.3 Languages and frameworks

The Genomic HyperBrowser

Vederhus' tools were written using the Genomic HyperBrowser (Vederhus, 2013), and the tools presented in this thesis extends and improves his tools. Therefore it was natural to write the tools in the same framework.

Python

The different tools presented here were written in Python. Python is a high-level programming language with simple, clean language and great flexibility. Python has dynamic typing and is an interpreted (not compiled) language, and it has an extensive standard library that is free to use. This makes it ideal for situations where fast development is important (Langtangen, 2009). In addition, the Genomic HyperBrowser framework is written in Python (Sandve et al., 2010), and Vederhus wrote his tool in Python (Vederhus, 2013).

A weakness with Python compared to compiled languages like C and C++ is that it is not very fast, but Python is a perfect "glue language", making it easy to move computational expensive operations to other programming languages, for instance C (Langtangen, 2009).

In this thesis, the user interface and the HyperBrowser specific parts of the tools were written in Python, while the computationally expensive parts was moved to C. The HyperBrowser statistic will only gather the necessary data before calling the C code that calculates the FET and CSS analyses.

C

As mentioned above, the computational expensive parts was written in C. C is a general-purpose programming language, central in the UNIX system. The UNIX system and many of the programs on it is written in C (Kernighan & Ritchie, 1988). C is a typed language, with several fundamental types: characters, integers and floating point numbers of several sizes. Since C is a more basic langauge, C deals with characters, numbers and addresses, and there is no way to deal directly with composite objects like lists and strings.

Unlike Python, C is a compiled language and the code has to be compiled to machine code before it can be run. Compiled languages generally run faster than interpreted languages, since the interpreted code has to be converted to machine instructions at runtime. C is a relatively low level language, and gives longer development time than Python and is somewhat more difficult to use. The big advantage with C is that it gives faster code. A lot of runtime can be saved by developing efficiently, by optimizing parts of the code, and using profiling and compiler optimizations.

In the implementation of the C code, we have used the following methods and tools:

- General code optimizations (see section 4.8)
- Compiler optimizations
- Profiling with gprof (Graham, Kessler & Mckusick, 1982)
- Leak checking with Valgrind (Seward & Nethercote, 2005)

Cython

Cython is a language based on Python, with some additional syntax for allowing type declaration (Behnel, Bradshaw & Seljebotn, 2009). Cython code has familiar syntax for Python developers, and is able to call external C libraries. The Cython code is translated into optimized C/C++ code, but keeps the interface of the original Python source code, making it possible to call a Cython program as a standard Python module. The Cython code in itself gives a significant speedup compared to regular Python, but it is its ability to call external C libraries that is its main strength for this application.

In this application, we used Cython to create a link between Python and C, making it possible for the HyperBrowser statistic to call the (parallel) C code.

4.1.4 Parallelization

In the previous sections several possibilities for parallelizing programs were discussed. The critical parts of the code, written in C, were parallelized. The Genomic HyperBrowser runs on an Abel node called Insilico. Abel is the computer cluster at the University of Oslo, theoretically capable of 258 TFLOPS/s peak performance. All regular compute nodes on the cluster have 16 physical

CPU cores, with minumim 64 GB RAM. Insilico is a special Abel node, used by the BMI for the Genomic HyperBrowser (and possibly other applications). It has 64 physical cores, four CPUs with 16 cores each. Each CPU is an AMD Opteron(TM) Processor 6276.

To be able to run the tools on Insilico, a parallel programming system created for shared-memory environments must be used. The choice fell on Pthreads, since it gives more control than OpenMP over the execution of each thread. With Pthreads, one has full control of the creation and termination of threads, and it is relatively easy to create threads, distribute tasks and data, and terminate threads after execution.

The tasks given to each thread were not of equal compuational size. Therefore we used the peer model, with a dynamic mapping of tasks. Each thread picks a new task, from a task pool, until there are no tasks left. This is futher described in section 4.5.

4.2 Data structure

The C program needs the following data:

```
void compute(double *avals, double *bvals, int *apos, int *bpos,
    int regstart, int regend, int wsize, int wstep,
    int alen, int blen, double *scores, double *stddev);
```

To make the code as fast as possible it is important that the Python part of the code do as little as possible. Therefore we tried to keep the data structures as they are.

- `avals` and `bvals`: Double arrays with the SNP values for population A and B. The values are ordered by position and then individual. The values for non-frequency data are 3 (major allele), -3 (minor allele), 0 and -10000 (no data).
- `apos` and `bpos`: The different positions in the genome are stored in these arrays, ordered by position and then individual:

  $$\{262, 262, 262, ..., 300, 300, 300, ...\}$$

  The elements in the two arrays `avals` and `apos` correspond, so that `avals[0]` corresponds to the value of an individual at position `apos[0]`.
- `regstart`: The start position of the region.
- `regend`: The end of the region, i.e. the last position in the genome
- `wsize`: The size of a window, default 2500.
- `wstep`: The step size, default 500.

- `alen and blen`: The length of the respective arrays for population A and B.
- `scores` and `stddev` / `p`: The double arrays where the scores (and standard deviation or *p*-value) for each window are stored. The index in the array corresponds to the start position of the window, divided by the step size.

The **Cluster Separation Score** has some additional arguments, to control the Monte Carlo test and choice of MDS algorithm:

- `treshold`: Minimum number of significance runs, default 10.
- `runs`: Maximum number of significance runs, default 200000.
- `drosophila`: The choice of distance metric: count differences or average of frequency.
- `mds`: The choice of MDS algorithm; Classical MDS (CMDS), SMACOF or a combination of the two.

In addition, the **Fisher Exact SNP Tool** needs the percentile score to select in each window:

- `perc`: The percentile of the L_{10}FET score selected for each window, default 0.95.

4.2.1 File format

The file format is the same that Vederhus (2013) used, with one important exception. The value type for data with several individuals, like the stickleback data, was changed from categorical to numeric:

```
##gtrack version: 1.0
##track type: valued points
##value type: number
###seqid start value genomeid
####genome=gasAcu1
chrV 262 3 12
...
```

With this change, the SNP values are stored as Python floats (`float64`). These floats can then be sent directly to the C module, without any numerical conversion, and this saved us some runtime.

The SNP values for data with several individuals are

- 3: Major allele
- -3: Minor allele
- 0: Both allele observed in reads

- -10000: No data

while the SNP values for pooled populations (like the *Drosophila* data) are the frequency of the minor allele. This is unchanged from Vederhus' thesis (Vederhus, 2013).

4.2.2 A tool for converting VCF to GTrack

A tool for converting a VCF file containing SNPs to a GTrack file was created. This tool was implemented as a pure Python tool in the Genomic HyperBrowser. Some assumptions were made of the VCF file:

- The individuals are listed in the header.
- Diploid calls are made.
- There are only two alleles per SNP, one reference allele and one alternate allele.

4.2.3 Sliding windows

Each of the following methods were implemented by sliding windows. The different scores are calculated for one window at a time, and slided a given step size along the genome before the next computation. The goal is to find the windows with the largest genomic divergence. The window size and step size are decided by the user. The data are stored in contiguous one dimensional arrays, and since we only have the SNP positions, there are "holes" in the genome. To find the array elements corresponding to a given window, we need to search through the position array, and find the correct start (`left`) and stop (`right`) indices in the array.

This method is based on the `SlidingWindow` data structure made by Vederhus (2013). To avoid searching from the beginning of the array each time, the indices `left` and `right` contains the previous `left` and `right` indices before the loops:

```
/* find the new array index position
of left and right idx-pointer */
while (left < length && positions[left] < start) {
    left++;
}

while(right < length && positions[right] <= stop) {
    right++;
}
```

4.3 Fisher's Exact Test

The **Fisher Exact Test SNP Tool** is available as a web tool in the comparative branch of the Genomic HyperBrowser. The code for this web tool can be found in `quick.webtools.restricted.FisherExactTestSNPTool`. The main features of this web tool is the same as Vederhus used for his thesis (Vederhus, 2013), but we have made some small changes, detailed in section 5.1. The real changes were made in the statistic, which is described in section 4.3.2. We have included a short description of the web tool, for better understanding of how the tool works.

4.3.1 The web tool

The user selects the different parameters of the analysis: genome build, files from history, window and step size, percentile score in window and output format (tabular or html). The chosen files are then pre-processed by the HyperBrowser system, by the following calls:

```
tn1 = ExternalTrackManager.getPreProcessedTrackFrom\
    GalaxyTN(genome, track1);
tn2 = ExternalTrackManager.getPreProcessedTrackFrom\
    GalaxyTN(genome, track2);
```

where `track1` and `track2` are the files chosen by the user. The pre-processing is only performed the first time the analyses are run on new input files, and is very slow. For large data sets, this can take several hours. The computations start in the `runManual` method in `GalaxyInterface`. Here, the analysis is run, one chromosome at a time:

```
result = GalaxyInterface.runManual([track1, track2], \
    analysisDef, reg, bins, genome, galaxyFn=galaxyFn);
```

The output from this function is written, for each window, to a file. This file is the output from the tool.

4.3.2 The statistic

When `runManual` is called, a chain of events starts, more detailed explained in section 2.4, which ends with the creation of the relevant `Statistic`. The statistic created for the FET tool is `FisherExactScoreStat` in `quick.statistic`. Here, the relevant data are fetched, and the Cython method `fisher_exact_tester` in `quick.statistic.fisher_cython` is called:

```
fisher_exact_tester(avals, bvals, apos, bpos, regstart, regend,\
    wSize, wStep, alen, blen, perc, scores, stddev);
```

This is a Cython wrapper method, that converts the data to the necessary format, and calls the C program:

```
def fisher_exact_tester(np.ndarray[np.float64_t, ndim=1] avals, \
..., np.ndarray[np.float64_t,ndim=1] stddev):
    compute(<double*> avals.data, ..., <double*> stddev.data);
```

Futher description of the Cython code is given in section 4.7.

In the C part of the program, the different windows are slided through. For each window, the negative $\log_{10}$ FET score ($\mathrm{L_{10}FET}$) is computed for each SNP position. Then, a $\mathrm{L_{10}FET}_{(1-x)\%\mathrm{Q}}$ score is computed for each window, which is the $x\%$ percentile of the scores in the window, with x given by the user. The standard deviation of the scores in the window is also returned:

```
/* For each SNP position, count frequencies and
calculate the negative log 10 fet score */
for (i = npos; i--; ) {
    fetcount(f, avals, bvals, i, asize, bsize);
    fetscore = fet(f, tmp);
    fetscores[i] = -1.0*log10(fetscore);
}

/* calculate the percentile */
results[0] = percentile(fetscores, npos, perc);

/* calculate the std.dev. */
results[1] = calc_std(fetscores, samples,
    stdsamples, nsamples, npos, perc, state);
```

There are four important functions here: `fetcount`, `fet`, `percentile` and `calc_std`.

`fetcount` counts the number of occurences of the minor and major allele in the two populations. For the stickleback data, it creates the contingency table:

	Freshwater	Saltwater	
Major	a	b	$R_1 = a + b$
Minor	c	d	$R_2 = c + d$
	$C_1 = a + c$	$C_2 = b + d$	$N = a + b + c + d$

`fet` computes the two-tailed Fisher's Exact Test from the contingency table. It uses the formula proposed and explained by Feldman and Klinger (1963) and Zar (1987). The most important function here is `fet_p` that computes the p-value for the table, with the binomial formula described in section 3.2:

```
nom = binomial(ab, a) * binomial(cd, c);
denom = binomial(n, ac);
return nom/denom;
```

`percentile` This method sorts the scores, and computes the $x\%$ percentile:

```
qsort(fetscores, n, sizeof(double), compare_doubles);
idx = (n-1)*percentile; // indices from 0 to n-1
delta = (n-1)*percentile - idx;
return (1-delta)*fetscores[idx] + delta*fetscores[idx+1];
```

`calc_std` This method computes the standard devaition of n bootstrap replicate samples of the $\mathrm{L_{10}FET}_{(1-x)\%\mathrm{Q}}$ score. This is done by selecting `npos` samples from the scores in the window by bootstrap sampling, and calculating the $x\%$ percentile of this selection. Finally, the standard deviation of these n scores is calculated and returned:

```
for (i = 0; i < nsamples; i++) {
    bootstrap_sample(fetscores, samples, npos, npos, state);
    stdsamples[i] = percentile(samples, npos, perc);
}
return std(stdsamples, nsamples);
```

The bootstrap samples are selected randomly using the the C library function `nrand48`; a fast, thread safe random number generator. A more detailed description is given in section 4.6. The `bootstrap_sample` function is given below:

```
for (i = n; i--; ) {
    idx = random_int_nrand48(npos, state);
    samples[i] = fetscores[idx];
}
```

The results are stored in two large double arrays, `scores` and `stddev`, and returned to the statistic. From the statistic, the results are returned back to the web tool, where they are written to the output file.

To filter the scores, the user needs to use Vederhus' tool `FilterFisherScores` in `quick.webtools.restricted`. See (Vederhus, 2013) and section 5.1.2 for more information about this tool.

4.4 Cluster Separation Score

This tool works in the same way as the Fisher's Exact Test tool. The different tools and statistics can be found at

- The web tool: `quick.webtools.restricted.ClusterSeparationScore`
- The statistic: `quick.statistic.CategoryClusterSeparationStat`
- The Cython module: `quick.statistic.css_cython` with the function `cluster_separation_scorer`.

The Cython wrapper function `cluster_separation_scorer` takes three additional inputs compared to the FET module, all integer values. Two of them, `threshold` and `runs`, control the Monte Carlo test, while the third, `drosophila`, takes on the values 0 and 1 and decides which distance metric that should be used:

```
cluster_separation_scorer(avals, bvals, ..., threshold,
    runs, drosophila, scores, p);
```

The core of the CSS method, computed for each window, is the two functions `cluster_separation_scorer` and `significance_treshold`. The first function computes the Cluster Separation Score, while the other estimates the p-value with a Monte Carlo test.

The cluster separation scorer consist of three main steps:

- Compute the dissimilarity matrix Δ, using a suitable distance metric.

```
/* 1: Pairwise compare all individuals with
a distance measure */
if (drosophila) {
    compare_freq(avals, bvals, npos, dissimilarity);
} else {
    compare_all(avals, bvals, asize, bsize, npos,
            dissimilarity);
}
```

- Scale the data down to two dimensions using a MDS method, for instance classical MDS:

```
/* 2: Use mds to scale down to two dimensions */
if (mds == 0) {
    /* Classical MDS */
    cmds(dissimilarity, X, dims, m, B, Z, tmp, L, Q);
}
```

- Compute the CSS, using the formula given by Jones et al. (2012). The distances are calculated once per window:

```
/* 3: Calculate the cluster separation score
precalc. the distance */
calc_dist(X, distance, m);
return css(distance, atracks, btracks, asize, bsize);
```

4.4.1 Distance metrics

Vederhus developed two distance metrics (Vederhus, 2013), count differences and average of differences. For the stickleback data he used count differences, which is the number of positions in the window where the set of values is $[3, -3]$. The generation of the dissimilarity matrix consists of three loops, two for comparisons whitin each group and one for comparisons between the two groups. The code for comparing the individuals in group A with itself is seen below:

```
/* for comparing avals with it self */
for (i = asize; i--; ) {
    for (j = i; j--; ) {
        count = 0;
        for (k = npos; k--; ) {
            if (avals[k*asize + i]*avals[k*asize + j] == -9) {
                count++;
            }
        }
        dissimilarity[i][j] = count;
        dissimilarity[j][i] = count;
    }
}
```

Since the dissimilarity matrix is symmetric, only half of the matrix needs to be calculated. The diagonal is not computed, since the dissimilarity between to equal objects is 0.

For pooled data, like the *Drosophila* data, we have the population frequency of the minor allele for each SNP position.

$$avals = [0.75, 0.33, 0.22, 0.3, ...]$$

Vederhus (2013) therefore used the average of the absolute values of allele differences as a distance metric. Since the frequency data only give one value per population, the dissimilarity matrix is a 2×2 matrix. The method for creating the dissimilarity matrix therefore becomes quite short and fast:

```
for (i = npos; i--; ) {
    avg += dabs(avals[i] - bvals[i]);
}
avg /= npos;

dissimilarity[0][1] = avg;
dissimilarity[1][0] = avg;
```

Here, the method `dabs` is used, which gives the absolute value as a double.

Jones et al. (2012) filled the empty elements of the matrix ($\delta_{ij} = 0$) with the average value of the matrix before the MDS. They also discarded windows with

too few comparisons. We have chosen to do something similar; when over half of the dissimilarity matrix is equal to 0, the window is discarded. The code for this can be found in `fill_averages`.

4.4.2 MDS methods

Classical MDS

The method `cmds` implements the classical MDS and uses several methods from the GNU software library (Galassi et al., 2009). The library is used for matrix multiplication:

```
gsl_matrix_view a = gsl_matrix_view_array(A[0], m, n);
gsl_matrix_view b = gsl_matrix_view_array(B[0], n, p);
gsl_matrix_view c = gsl_matrix_view_array(C[0], m, p);

/* Compute C = A B */
gsl_blas_dgemm (CblasNoTrans, CblasNoTrans,
           1.0, &a.matrix, &b.matrix,
           0.0, &c.matrix);
}
```

and for computing the eigenvalue decomposition $B_\Delta = Q\Lambda Q^T$.

SMACOF

The main loop of the SMACOF algorithm for calculating MDS is shown below:

```
while (k == 0 ||
    ((sigma_prev - sigma) > epsilon && k <= max_iters)) {
    sigma_prev = sigma;

    // Step 4: increase k with 1
    k++;

    // Step 5: Compute the Guttman transform X^k
    guttman_transform(X, B, Z, D, dissimilarity, m, dims);

   // Step 6: sigma_r^k =sigma_r(X^k)
   calc_dist(X,D,m);
   sigma = stress(dissimilarity, D, m);

   // Step 7: Set Z = X^k
    copy_matrix(Z, X, m);
}
```

As we can see, the SMACOF method runs until convergence of the solution, when $(\sigma_{prev} - \sigma) < \epsilon$, or when the number of iterations have reached `max_iters`

iterations. To increase the chance of a global optimum, the algorithm is run with several different random start points, and the best solution is selected. The default method is run with 4 different random start points, and the maximum number of iterations is 300 and $\epsilon = 10^{-6}$.

Combination of SMACOF and CMDS

In the combination algorithm, the solution **X** from the CMDS function is used as the start point to the SMACOF algorithm:

```
cmds(dissimilarity, X, dims, m, B, Z, tmp, L, Q);
smacof(dissimilarity, m, dims, X, Q, B, Z, max_iters, epsilon);
```

4.4.3 CSS

To calculate the CSS the Euclidean distances between all individuals are required. This is calculated once per window:

```
for (i = m; i--; ) {
    for (j = i; j--; ) {
        ans = sqrt((A[i][0] - A[j][0])*(A[i][0] - A[j][0]) +
        (A[i][1] - A[j][1])*(A[i][1] - A[j][1]));
        distance[i][j] = ans;
        distance[j][i] = ans;
    }
}
```

The distance matrix is symmetric, therefore only half of the matrix is calculated. The function `sqrt` from the C math library is used. The CSS is calculated directly from the formula described in section 3.1. A part of the `css` method is shown below, the calculation of the distance between the two populations:

```
//average between-group distance
bet_dist = 0;
for (i = asize; i--; ) {
    for (j = bsize; j--; ) {
        bet_dist += distance[atracks[i]][btracks[j]];
    }
}
bet_dist = bet_dist/(asize*bsize);
```

The two integer arrays `atracks` and `btracks` contain the individual IDs for the two populations. The IDs in population A run from 0 to $asize - 1$, while the IDs in population B run from $asize$ to $asize + bsize$. With the help of these two arrays, the correct indices in the Euclidean distance matrix can be obtained, even when the population groups are mixed up. This is necessary for estimating the significance.

4.4.4 Estimating significance

The significance of the CSS is estimated with a Monte Carlo test, just as in (Vederhus, 2013). The main loop of the function `significance_threshold` is given below:

```
while (hits < treshold && nscores < runs) {
    random_shuffle(tracks, ntracks, state);
    atracks = &(tracks[0]);
    btracks = &(tracks[asize]);

    newscore = css(distance, atracks, btracks, asize, bsize);
    if (newscore >= score) {
        hits++;
    }
    nscores++;
}
```

The parameters `threshold` controls the minimum number of hits we need and `runs` controls the maximum number of iterations.

All individuals are represented by their individual IDs, and the integer array `tracks` contains all the IDs for the individuals in both populations. By shuffling the elements of this array, a new division of the individuals into two groups is obtained. The array is shuffled by a Fisher-Yates shuffle (Durstenfeld, 1964), (Knuth, 1997):

```
for (i = n-1; i > 0; i--) {
    r = random_int_nrand48(i+1, state);
    swap(&elms[i], &elms[r]);
}
```

In the Fisher-Yates shuffle, a pseduo-random number generator is called as many times as the number of individuals. At worst, this shuffle method is called several hundred thousand times per window. We therefore needed a fast random number generator. For this, we used the C library pseduo-random generator `nrand48`, see section 4.6. Both the `random_int_nrand48` and the `swap` function are inline functions, for increased speed.

The p-value is estimated as $p = (r + 1)/(n + 1)$:

```
p = (hits+1)*1.0/(nscores+1);
```

For the p-value to be good enough, n has to be a large number and r has to be at least 10 (North, Curtis & Sham, 2002). Therefore the default values for the minimum amount of hits needed and the number of iterations are 10 and 200 000, respectively.

4.5 Parallel implementation

The calulations were parallelized for increased speed. The smallest possible task is a single window, and since each window is independent, there are no communications between threads during computation. All windows are of equal length, but the computational size of each window is not equal, since there are an unknown amount of SNPs in each window. In addition to this, the calculation time for the CSS tool varies a lot between windows, due to the Monte Carlo test for estimating p-value, and the variable runtime of the SMACOF algorithm for MDS.

Dividing the chromosome into n tasks of equal size will not work well for this application, since the windows are not equally expensive. Therefore we kept the task size constaint across all chromosomes, with each task having x windows. Some windows will be more expensive than others, and become bottlenecks for the parallel program. By keeping the tasks as small as possible, we hoped to minimize these bottlenecks, by dividing the heavy computations between several threads. Too small tasks, on the other hand, will give unnecessary overhead in the program. Therefore we had to find a balance between task size and overhead. The runtimes for several different task sizes can be found in section 5.4.1.

We chose to use the peer model to map the tasks to the different threads, with a dynamic mapping of tasks. One thread, the main thread, starts all the other threads, and then waits for them to finish. The other threads select new tasks from a common thread pool until there are no more tasks. They are then joined and terminated by the main thread. In this way we hoped to minimize the bottleneck of the computationally expensive tasks.

4.5.1 Pthreads implementation

The parallel code was implemented with Pthreads. The 'main' thread creates the other threads, and sends them the necessary data with the help of the struct `thread_data`, here shown for the CSS tool:

```
struct thread_data{
    int thread_id;
    int num_windows;
    ...
    double *scores;
    double *p;
};
```

A lot of data is needed to start our program. The threads are started with the following code:

```
if ((rtn = pthread_create(&threads[i], NULL, (void*)mycompute,
  (void*)&thread_data_array[i]))) {
    printf("Error: pthread_create %d, %s\n", i, strerror(rtn));
    exit(-1);
```

```
}
```

The computation for each thread starts in the method `mycompute`. The threads have one global shared variable, the variable `task_id`, which contains the number of the next available task. To avoid race conditions, this variable is protected by a global mutex variable

```
pthread_mutex_t mutexTASK_ID;
```

When a thread needs to access the variable `task_id`, it first has to aquire the lock of the mutex variable `mutexTASK_ID`. When this lock is aquired, all the other threads must wait in line for the lock to be freed, and the thread can safely select a new task by modifying the variable `task_id`. When the thread is done with the variable, the mutex lock is released, giving the other threads access. This is done by the following code:

```
pthread_mutex_lock (&mutexTASK_ID);
my_task_id = task_id;
task_id++;
pthread_mutex_unlock(&mutexTASK_ID);
```

The necessary data for the new task are fetched by a call to the method `get_positions`, which gives the corresponding array positions of the task. `tid` contains the task id:

```
/* start value */
l = (tid*TASK_SIZE)*wstep;
*start = l;

/* end value */
r = ((tid+1)*TASK_SIZE)*wstep + (wsize - wstep);
*stop = r;
```

The result for a given window is written to the two result arrays with the correct index. All threads allocate their own data needed for the calculations, such as the matrices needed for MDS, but they share the same input and result arrays. Since they work at different positions, linked to a given task id, there are no race conditions.

When there are no more available tasks, the threads exit, and are joined by the waiting main thread:

```
for (i = 0; i < NUM_THREADS; i++) {
    if ((rtn = pthread_join(threads[i], NULL))) {
        printf("Error: pthread_join %d, %s\n", i, strerror(rtn));
        exit(-1);
    }
}
```

Several pseudo-random number generators (PRNGs) are used in the programs. To avoid problems, each thread has its own stream of random numbers. This is achieved by letting each thread have its own seed, that is sent to the PRNG. This is explained in detail in section 4.6.

4.6 Pseudo-random number generators (PRNGs)

The tools presented here need a fast, reliable pseudo-random number generator. The CSS uses a Monte Carlo test to estimate significance, and the FET uses bootstrap sampling to generate the standard deviation in the window. For both these applications we needed a fast generator. Both tools uses Pthreads for parallelization, so the PRNG needs to be thread safe, without being a bottleneck.

4.6.1 A thread safe pseudo-random number generator

A computer will not produce true random numbers, but it can produce numbers that have statistical properties of randomness (Knuth, 1997). The functions that generates these numbers are called pseudo-random number generators (PRNGs).

A PRNG is a kind of finite state machine. It is defined by three properties:

1. An initial state, s_0.
2. A transition function $s_{k+1} = S(s_k)$ that transfers the machine from one state to another.
3. An output function, $V(s_k)$ that computes a random number from the current state.

The transition function and the initial state defines the sequence of pseudo-random numbers. Using the same initial state (or *seed*) produces the same sequence of numbers.

The `random()` function in the C library works in this way; the initial state is set by sending a seed to the function `srandom()`. Each time `random()` is called it reads the current state, creates a random number, and updates the old state.

When working with threads, shared states create problems. If the shared state is not protected, problems like race conditions can occur, where one thread reads the state while another thread is modifying it. To avoid race conditions, the state can be protected, by making sure state read and writes are only performed by one thread at a time. This can be achieved by giving each thread mutually exclusive access to the shared state. Then the PRNG will become a bottleneck, and the resulting program can end up almost sequential. In our first version of the parallel program, `random()` was used to generate pseudo-random numbers with several threads, and the run time of the program more than doubled compared to the serial version. This is clearly not good enough.

A better solution, that ensures a thread safe PRNG without creating additional overhead, is to let each thread have its own state and stream of numbers.

Race conditions will then be avoided, and a thread safe PRNG is gained without unnecessary overhead. The standard C library function `nrand48()` was a good solution to our problem; it is a thread safe PRNG that lets each thread have its own state. The transition function used is the linear congruential formula (Knuth, 1997):

$$X_{n+1} = (aX_n + c) \mod m, \ \text{n} \ \geq 0$$

For `nrand48()`, $m = 2^{48}$ and this creates a sequence of 48-bit integers, X_i. This formula is fast and efficient, and the sequence should be large enough for our purpose.

4.6.2 Uniform distribution of pseudo-random numbers

The following code will not give a perfectly uniform distribution of the random numbers between 0 and $n-1$:

```
nrand48() % n
```

It is possible to get an uniform distribution by modifying the code a little, by fetching a new random number if the generated number is too high. We used the following function for the PRNG, adapted from the VSEARCH function `shuffle.random_int` (Rognes, Mahé & xflouris, 2015):

```
long random_max = RAND_MAX;
long limit = random_max - (random_max + 1) % n;
long r = nrand48(state);
while (r > limit)
    r = nrand48(state);
return r % n;
```

This function will give uniformly distributed pseudo-random numbers without costing to much runtime.

4.7 Integrating C code with Cython

We used Cython to link the C program with the Python code. The Cython program is stored in a `.pyx` file. In this file, the methods used from the parallel C program must be defined. Each C header file must be included, and header files without methods we call directly can be defined as follows:

```
cdef extern from "cFisher.h":
    pass
```

while the methods we are going to call directly must be specified:

```
cdef extern from "threadfisher.h":
     void threadcompute(double *avals, ..., int regstart, ...)
     pass
```

Notice that one needs to use types for the variables, just as in C. `cdef` defines a C-function that is only callable from Cython. To be able to call the method from a Python program, a wrapper method must be written:

```
def fisher_exact_tester(np.ndarray[np.float64_t, ndim=1] avals, \
..., int regstart, ...):
     threadcompute(<double*> avals.data, ..., regstart, ...)
```

The `numpy` array parameter must be specified as:

```
np.ndarray[np.type, ndim=1] arrayname
```

and before the array is sent to the C method, it must be cast to the right type:

```
<type*> arrayname.data
```

The single variables, such as the integer variable `regname`, can be sent as they are.

Cython programs must be compiled. The compilation is done in two steps: First the `.pyx` file is compiled to a `.c` file by Cython. Then, the `.c` file, together with the necessary C code files are compiled and linked to a `.so` file by a regular C compiler. This can easily be done in one step, with the use of a distutils `setup.py` (Behnel et al., 2009), or with a customized bash script.

To use the Cython program from a regular Python program, the corresponding `.so` file is needed. Then, the program can be imported like a regular Python module:

```
from fisher_cython_parallel import fisher_exact_tester;
```

and called like any other Python function:

```
fisher_exact_tester(avals, bvals, apos, bpos, regstart, \
    regend, wsize, wstep, alen, blen, perc, scores, stddev);
```

4.7.1 Problems with integrating C code in a large scale Python system

Cython is a tool that makes it easy to call C code from Python code. However, integrating C code can be a difficult issue on a large, complex system, and we ran into several issues trying to integrate the C code with the HyperBrowser system.

Python comes with different representations of Unicode characters. The standard is ucs2, which corresponds to UTF-16, a 16-bit encoding of characters.

The HyperBrowser uses this version. Most Linux distributions, on the other hand, ships Python with ucs4, which corresponds to UTF-32. Therefore we had to compile the Cython code on Insilico.

The standard/automatic way of compiling the Cython code, with distutils (Behnel et al., 2009), did not work on Insilico, since it uses the Inter C compiler (icc) instead of the GNU C compiler (gcc). The solution to this problem was to manually compile and link the Cython code. This was done using a bash script, `fisher_parallel_setup.sh`:

```
cython -a fisher_cython_parallel.pyx

icc -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -xAVX\
-mavx -fPIC -I/cluster/software/VERSIONS/python2-2.7.3/inc\
lude/python2.7 -c fisher_cython_parallel.c -o \
build/fisher_cython_parallel.o
...

icc -pthread -shared -O3 -xAVX -mavx build/fisher_cython_parallel.o\
build/cFisher.o build/comparative.o build/threadfisher.o \
-L/cluster/software/VERSIONS/python2-2.7.3/lib -lm -lpython2.7 \
-o /hyperbrowser/src/hb_core_comparative/quick/\
statistic/fisher_cython_parallel.so
```

In this script the Cython code, stored in `fisher_cython_parallel.pyx`, is compiled, together with the C source code. The resulting `*.o` files are stored in the directory build. After the compilation is done, all the files are linked into the module `fisher_cython_parallel.so`. This is the module the method `fisher_exact_tester` is later imported from. Notice that the files are linked with the C math library (-lm) and the Pthread library (-pthread).

With this script the Cython and C code can be compiled:

```
bash -x fisher_parallel_setup.sh
```

and later called from the HyperBrowser tool, as shown above.

From this, several important lessons was learned, for instance that it is important to compile the code where it is going to be used, and that it is important to ask for help early in the process.

The CSS Cython code was compiled and linked in a similar manner.

4.8 Optimizing C code

To further speed up the (serial) C code, code optimizations are important. They can improve the run time of a program, but a lot of optimization can also be done by the compiler with compiler options, like the flag `-O3`. It is important, however, not to optimize too much or too soon; this can break otherwise functioning code, and make the code completely unreadable. Optimizing non-critical parts of the code are a waste of time, as noted by Knuth (1974):

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." (p. 268)

With both manual and compiler optimization, and the use of profiling tools such as Valgrind (Seward & Nethercote, 2005) and gprof (Graham et al., 1982), we were able to speed up the (serial) C program.

4.8.1 Allocations

It is important to avoid unnecessary allocations and frees. The MDS methods need a lot of helper matrices. These matrices are allocated once per thread and are sent to the relevant method. In this way, a lot of unnecessary allocations and frees are avoided. By moving the allocations outside the loop, and only doing one allocation per chromosome, we saved a lot of runtime, at the cost of slightly more complicated code, and more data sent to each method, as can be seen in the code examples in section 4.3 and 4.4.

4.8.2 Library methods

It is often wise to use library methods. We saved some runtime by using GNU software library (Galassi et al., 2009) methods for matrix multiplication and the calculation of eigenvalues and -vectors, instead of writing our own methods.

4.8.3 Pseudo-random number generators

As discussed in section 4.6 we needed a fast and reliable PRNG for our program. Both the CSS and FET need to generate many random numbers per window. At worst, the CSS needs several hundred thousand random numbers. In these situations it is important to use a fast generator. As discussed in section 4.6, it is important that the PRNG is thead safe without creating overhead, so each thread should have its own stream of numbers. It is important that the seed is only generated once per thread.

4.8.4 For-loops

For short loops, it can be useful to write out the code instead, so called loop-unrooling. This makes the code longer and and a bit harder to read, but speeds up the program. It is also important to avoid too many function calls inside loops.

4.8.5 Data types

In C, integer arithmetic is faster. Therefore the use of char, short and double should be avoided, and int should be used instead, if possible. Unfortunately, in this application most of the data are stored as floats in the HyperBrowser system. The cost of converting the data in Python is much larger than the

additional (small) cost of double arithmetic in C, so most of our data are stored as doubles.

4.8.6 Functions

Some inline functions are used in critical parts of the code. We have also tried to avoid uncessesary function calls in loops. This makes the code a bit longer, but avoid the unnecessary overhead of calling a small function many times.

4.9 Changes made to the HyperBrowser code

The comparative genomics tools presented in this thesis reside on a separate branch of the HyperBrowser: https://hyperbrowser.uio.no/comparative

Vederhus' tool (Vederhus, 2013) was written on the then current version of the HyperBrowser, now outdated. This version did not handle pre-processing of large data files. Near the end of this thesis, we therefore merged in the new version of the HyperBrowser, to be able to work with new code and pre-process the large stickleback data. To make this work with our existing code, we did two important changes:

The code in the CSS tool requires that the elements are sorted in correct order, by position first, and then genomeid. We therefore changed the sorting algorithm used by the pre-processing method to merge sort, which is guaranteed to leave the order of the genomeid as it was. This change is done in `gold.origdata.OutputFile` in the method `sort()`:

```
sortOrder = self._contents.argsort(kind='mergesort')
```

The method `_removeBlindPassengersFromNumpyArray` in `gold.track.TrackView` removes elements that does not belong in the array (so called stowaways) for data given in intervals. This method is not needed by these tools, since the data are given as points and not intervals, and it uses up a lot of the run time. Therefore, we have added some small changes to this method to be able to skip this calculation for point data:

```
#Can not be blind passengers if length of elements is always 1..
if (not self.trackFormat.isInterval()) or \
(self.genomeAnchor.start==0):
    return numpyArray
```

These two changes have to be dealt with if the comparative branch is going to be merged into the standard branch of the Genomic HyperBrowser.

Chapter 5

Results

In this thesis we present new and improved versions of the tools made by Vederhus (2013). The tools are improved in important aspects: the code for calculating the statistical analyses are changed to improve the run time, and the user interface is greatly improved. The changes in the tools, together with the results from the analyses on several different data sets, are presented in the following sections.

5.1 User interface

Below is a list of the complete HyperBrowser tools for comparative genomics:

- **Fisher Exact Test SNP Tool**
- **Filter Fisher Scores**
- **Cluster Separation Score**
- **Significant CSS Regions**
- **Convert Stickleback Snps to Gtrack**
- **Convert VCF To Gtrack Tool**

All tools are available on the comparative branch of the Genomic HyperBrowser: https://hyperbrowser.uio.no/comparative, under a header named 'Comparative Genomics'. An example run of the new tools can be found in Appendix A.1, and descriptions of the original tools can be found in Vederhus (2013). There are also short descriptions at the bottom of each tool web page, some written by Vederhus and some written by us, together with a link to an example history where the tool is used.

As written more detailed in section 4.1, we have made large changes to the statistics of the **Fisher Exact Test SNP Tool** and the **Cluster Separation Score**. In this section the changes of the user interface are presented.

The largest change in the user interface is that the tools can now be run for the entire genome, instead of each chromosome separately, saving a lot of manual work. The tools need two GTrack files per genome, one for each population. This was achieved by merging the codebase with the new stable HyperBrowser code. With this new code, the tools are able to pre-process large data files, thus being able to run for the entire genome at once.

For each tool, the 'html' option is changed to 'customhtml', so the user can get intelligible results by clicking on the eye.

5.1.1 Fisher Exact Test SNP Tool

The FET tool calculates a FET score and a corresponding standard deviation for each window in the genome. The results for each window are written to an output file. As mentioned above, this tool can now be run for the whole genome at once. We have made some changes to the user interface by letting the user select what percentile L_{10}FET score should represent the window. This is selected in the field 'Percentile of L10FET scores in a window', that can be seen in figure A.3 in Appendix A.1.2. The default value here is 0.95, which is the value used by Burke et al. (2010). We have also added a description of the tool on the tool web site, to make it easier to use, together with links to example runs.

5.1.2 Filter Fisher Scores

This tool filters out the relevant regions returned from the FET tool. We have changed this tool slightly, by:

- Changing the order of the user interface, with genome build at the top
- Changing the file selection to history file selection instead of check boxes. Since the results from the FET tool are stored in one file (instead of one file per chromosome), we no longer need to select several files at once
- Changing the header of the output GTrack file to 'sorted elements: false' instead of 'true'
- Letting the user set the percentile of the standard deviation over all windows. This value is used to filter out windows, by generating an appropriate limit. The default value is 75 (corresponding to the 75th percentile), which is the value used by Burke et al. (2010)

The web user interface can be seen in figure A.4 in Appendix A.1.2

5.1.3 Cluster Separation Score

The CSS tool calculates a cluster separation score and a corresponding p-value for each window in the genome. The results for each window are written to an output file. As for the FET tool, this tool can now be run for the entire genome

at once. We have not made any additional changes to the user interface of this tool.

5.1.4 Significant CSS Regions

This tool filters out relevant regions found by the CSS tool. The user has the choice of two filtering methods: to filter by the p-values with a given false discovery rate (FDR), or to filter the top x CSS regions. Normally, we would expect the user to filter by p-values, but in some cases it could be useful to filter by the top scoring CSS regions. CSS and p-value are generally highly correlated, but exceptions can occur (Jones et al., 2012). For frequency data like the *Drosophila* data set, we are not able to calculate any p-values, and we must therefore filter by the top scoring regions.

As for the Filter Fisher Scores tool, we have moved the different fields around, to make the user interface similar to the CSS tool, with the genome build at the top, and made the file selection simpler, with history files selection instead of check boxes. The filtered windows are combined to larger regions, but there was an error in the code that did this. According to Vederhus, the same method was used for combining filtered windows into larger regions for both filtering tools. Therefore, Vederhus' code for the Filter Fisher Scores tool was used here.

The web user interface of this tool can be found in figure A.5 in Appendix A.1.

5.1.5 Convert Stickleback SNPs to GTrack

Since we have made a small change to the original file format, see section 4.2.1, this tool was updated accordingly, by changing the header file. The user interface is unchanged, and can be seen in figure A.2 in Appendix A.1.

5.1.6 Convert VCF To GTrack Tool

This tool is brand new and converts VCF files to our custom GTrack format. This tool has to be run once per population. It needs a VCF file with SNPs and a list of the individuals in the population. We have made some assumptions about the VCF file, these can be found in section 4.2.2. The list of individuals can be given as a text file, with one individual on each line, or as a comma separated list in a text box. The user interface is shown in figure A.7 in Appendix A.2, together with an example run of the tool.

5.2 Speedup of serial C code vs Python code

In this section, the runtime of the old Python program, the tool made by Vederhus (2013) is compared with the new serial C code. The code is run on the stickleback data (Jones et al., 2012), the data set used by Vederhus that took the longest time to run. All analyses are run with the default parameters, the same parameters Vederhus ran his analyses with:

- window size = 2 500
- step size = 500
- treshold = 10
- runs = 200 000

The serial C code is run for the whole genome, while the Python code is run for each chromosome separately. Both analyses are run on the Genomic HyperBrowser. Since the HyperBrowser is a multi-user system, the runtime may vary with the total load of the system. The runtime for the Python code is taken from Vederhus' old histories. These can be found at (Vederhus, 2013):

- Cluster Separation Score:

  https://hyperbrowser.uio.no/comparative/u/torkilve/p/stickleback-css

- Fisher's Exact Test:

  https://hyperbrowser.uio.no/comparative/u/torkilve/p/stickleback-fet

Since the C code is run for the whole chromosome at once, while the Python code is run for one chromosome at a time, there can be some additional start up costs for each run that is not included in the runtime for the C program. But, the C code saves a lot of manual work, allowing the user to do one run instead of one per chromosome, so the comparison should be fair.

All the runtimes are given in seconds, and are rounded up to one decimal with standard rounding rules. Speedup is given by

$$\frac{T_{old}}{T_{new}}$$

5.2.1 Cluster Separation Score

In the following, the serial runtime of CSS with two different MDS methods, CMDS and SMACOF, are presented. Vederhus used the SMACOF method for calculating MDS. Vederhus did not supply data for chromosome IV, but we have chosen to present the results from all chromosomes for our C program. This means that the total runtime of the C program will include one chromosome more than the total runtime for the Python program, which will make the speedup different for the the average and total runtime.

CMDS method

The C program gains a speedup of about 774× on average compared to the Python program. The total speedup is somewhat lower, since we do not have chromosome IV from Vederhus' run. The use of a faster algorithm (CMDS) and a different programming language (C) give a large speedup. The speedup varies a lot for the different chromosomes, from 108.2× to 1450.5×. This could

be because the CMDS method, if the different amount of SNPs per window is disregarded, does about the same amount of work per window (matrix multiplication, finding eigenvectors and -values), while the SMACOF algorithm varies a lot between windows, running until convergence of the solution.

Chromosome	Python	C (CMDS)	Speedup
ChrI	163619.6	112.8	1450.5
ChrII	13372.5	15.0	891.5
ChrIII	5592.1	9.1	613.0
ChrIV	–	113.3	–
ChrV	5085.6	8.9	571.4
ChrVI	5505.5	9.3	592.0
ChrVII	30122.1	34.8	865.6
ChrVIII	19178.5	20.1	954.2
ChrIX	14115.4	18.7	754.8
ChrX	6432.9	10.0	643.3
ChrXI	48788.0	59.8	815.9
ChrXII	12775.5	18.1	108.2
ChrXIII	7456.3	11.2	665.7
ChrXIV	5665.3	9.6	590.1
ChrXV	5748.3	9.3	618.1
ChrXVI	7918.9	12.5	633.5
ChrXVII	4931.8	8.0	616.5
ChrXVIII	8651.5	11.5	752.3
ChrXIX	40922.5	39.2	1043.9
ChrXX	19251.0	26.5	726.5
ChrXXI	20510.2	44.6	459.9
ChrUn	24835.0	26.3	944.3
Overhead/startup costs	–	3.0	–
Total runtime	466679.5	631.6	739
Average	22222.8	28.7	774

Table 5.1: Python versus C for the serial cluster separation score (CSS) program with classical MDS. The analysis is run with the default parameters. The runtime is given in seconds.

SMACOF method

Vederhus used the SMACOF algorithm for calculating MDS, and both the C and Python program are run with the same parameters:

- Number of start points: 4
- Maximum number of iterations: 300
- Epsilon: $1e - 6$.

In table 5.2 we see that we get an average speedup of 81×. The speedup between the different chromosomes still varies a lot, but not as much as for the CMDS.

Chromosome	Python	C (SMACOF)	Speedup
ChrI	163619.6	460.9	355
ChrII	13372.5	306.5	43.6
ChrIII	5592.1	214.9	26
ChrIV	–	503.9	–
ChrV	5085.6	158.1	32.2
ChrVI	5505.5	228.5	24.1
ChrVII	30122.1	378.0	79.7
ChrVIII	19178.5	262.5	73.1
ChrIX	14115.4	280.9	50.3
ChrX	6432.9	214.9	29.9
ChrXI	48788.0	251.4	194.1
ChrXII	12775.5	233.7	54.7
ChrXIII	7456.3	258.8	28.8
ChrXIV	5665.3	195.7	28.9
ChrXV	5748.3	207.7	27.7
ChrXVI	7918.9	240.4	32.9
ChrXVII	4931.8	190.2	25.9
ChrXVIII	8651.5	211.2	41.0
ChrXIX	40922.5	284.8	143.7
ChrXX	19251.0	269.8	71.4
ChrXXI	20510.2	179.5	114.3
ChrUn	24835.0	490.8	50.6
Overhead/startup cost	–	2.4	–
Total runtime	466679.5	6025.3	78
Average	22222.8	273.9	81

Table 5.2: Python versus C for the serial cluster separation score (CSS) program with the SMACOF method for calculating MDS. The analysis is run with the default parameters. The runtime is given in seconds.

5.2.2 Fisher's Exact Test

The C program is run with the default parameters for the FET tool. As mentioned in section 3.2, Vederhus did not calculate the full two-tailed FET, and did not calculate the complete standard devation used by Burke et al., therefore the C program does a lot more work than the Python program.

In table 5.3 we can see that the total and average speedup is 21×, which gives us a considerable speedup compared to the Python program, even though the C program does more work. The speedup for the different chromosomes

varies between 19.9 and 23.8.

Chromosome	Python	C	Speedup
ChrI	470.1	22.1	21.3
ChrII	374.7	17.0	22.0
ChrIII	247.8	11.4	21.7
ChrIV	570.3	28.6	19.9
ChrV	188.3	8.7	21.6
ChrVI	248.6	11.0	22.6
ChrVII	493.6	24.4	20.2
ChrVIII	314.0	15.1	20.8
ChrIX	344.8	16.0	21.6
ChrX	249.2	11.4	21.9
ChrXI	278.7	13.4	20.8
ChrXII	291.6	13.6	21.4
ChrXIII	312.0	14.5	21.5
ChrXIV	221.6	10.0	22.2
ChrXV	230.9	10.1	22.9
ChrXVI	286.0	13.2	21.7
ChrXVII	226.7	10.0	22.7
ChrXVIII	241.3	10.9	22.1
ChrXIX	296.3	13.3	22.3
ChrXX	331.4	15.7	21.1
ChrXXI	211.2	10.4	20.3
ChrUn	762.6	32.0	23.8
Overhead/startup cost	–	2.7	–
Total runtime	7191.7	335.5	21.4
Average	326.9	15.3	21.4

Table 5.3: Python vs. C for the serial Fisher's exact test (FET) program. The analysis is run with the default parameters. The runtime is given in seconds.

5.3 Memory usage

The memory is measured with the linux commands `/usr/bin/time` and `top`. The memory measured is the maximum resident set size of the process during its lifetime. The results from `/usr/bin/time` are compared to the output of `top` to make sure the memory reported is correct. All memory are given in MB, and the memory use is averaged over 4 runs. The memory is measured for the C program, for both the serial and the parallel program, for three different chromosomes of the stickleback data. The parallel program uses 64 threads and a task size of 100.

The total memory used per chromosome for the serial program depends on three factors:

- The population size, m
- The number of SNPs in the chromosome, n
- The number of windows in the chromosome, w

The largest data structures are the input and result arrays sent to the method:

- Two double arrays `avals` and `bvals`: $8nm$ bytes
- Two integer arrays `apos` and `bpos`: $4nm$ bytes
- Two double result arrays `scores` and `p` / `stddev`: $16w$ bytes

These arrays take up most of the memory, and are the same for both the CSS and FET. Each method allocates some additional data structures, but the main amount of memory used for both serial programs should be (in bytes):

$$8nm + 4nm + 16w = 4(3nm + 4w) = O(nm + w)$$

m is typically small compared to n and w. The $16w$ factor is the result arrays returned from the method. In the HyperBrowser tool, these are saved for each chromosome, so the memory requirements might increase with $16w$ for each chromosome, depending on how the system stores the results.

5.3.1 Cluster Separation Score

For the CSS we have some additional data structures, constant for all chromosomes, and the most expensive of those are the $m \times m$ two-dimensional arrays used for the MDS methods. We need 5 matrices, and each of these take additional $8m^2$ bytes. In addition to this, we allocate some $m \times 2$ matrices and some m matrices, and some relatively small arrays, included as the constant C. The total amount of memory used by the CSS tool (in bytes) is therefore:

$$4(3nm + 4w + 10m^2 + 14m) + C$$

For chromosome XXI, $m = 21$, $n = 176\,230$ and $w = 23\,434$, so we should use about 44.8 MB.

For the parallel program, the threads share the input data, but each thread allocates the data structures needed for the computation. Therefore we would expect an increase in memory use for the parallel program, corresponding to the number of threads.

As can be seen in tables 5.4 and 5.5 the parallel program uses somewhat more memory than the serial program. Both the serial and the parallel program use about 60 % of the memory used by the relevant data files. The memory versus file size factor is bigger for the smallest chromosome, chromosome V. This is especially visible for the parallel program. This is due to the fact that the memory requirements for the calculations are the same for all chromosomes,

and only the size of the input data varies; therefore the smallest chromosome will allocate more memory relative to the size of the input files. There are more variations in the amount of memory used by the parallel program than by the serial program.

Chromosome	Averaged memory use	File 1	File 2	Mem/file size
chrV	40.0	31.3	30.4	0.65
chrXXI	45.0	39.5	38.2	0.58
chrUn	147.4	128.3	123.6	0.59

Table 5.4: Memory use, serial program: Cluster separation score with classical MDS for chromosomes V, XXI and Un. All analyses are run with the default parameters. The memory use is given in MB.

Chromosome	Averaged memory use	File 1	File 2	Mem/file size
chrV	44.1	31.3	30.4	0.71
chrXXI	48.0	39.5	38.2	0.61
chrUn	149.6	128.3	123.6	0.59

Table 5.5: Memory use, parallel program: Cluster separation score with classical MDS for chromosomes V, XXI and Un. All analyses are run with the default parameters. The memory use is given in MB.

5.3.2 Fisher's Exact Test

For the FET the number of SNPs in each window, x, is an additional factor. At worst, x is as big as the window size, and for large windows, this could be a considerable factor. The memory use for the serial program is therefore:

$$4(3nm + 4w + 4x) + C$$

For chromosome XXI, with a window size of 2.5 kb, we should use about 44.8 MB of memory. As for the CSS, the memory use is expected to increase for the parallel program.

The results given in tables 5.6 and 5.7 are similar to the results found by the CSS: The parallel program requires somewhat more memory that the serial program, and the memory use versus file size factor is about the same, around 0.6.

5.4 Parallel C code

In the following, the results for the parallel programs are shown. The analyses are run on the stickleback data set, since it is the largest and most time con-

Chromosome	Averaged memory use	File 1	File 2	Mem/file size
chrV	39.8	31.3	30.4	0.65
chrXXI	44.6	39.5	38.2	0.57
chrUn	147.1	128.3	123.6	0.58

Table 5.6: Memory use, serial program: Fisher's exact test for chromosomes V, XXI and Un. All analyses are run with the default parameters. The memory use is given in MB.

Chromosome	Averaged memory use	File 1	File 2	Mem/file size
chrV	41.5	31.3	30.4	0.67
chrXXI	46.6	39.5	38.2	0.60
chrUn	149.4	128.3	123.6	0.59

Table 5.7: Memory use, parallel program: Fisher's exact test for chromosomes V, XXI and U. All analyses are run with the default parameters. The memory use is given in MB.

suming data set we have [1]. We show how well the parallel program scales, with different amount of threads and different task sizes. All the analyses are run with the default parameters, given in section 5.2. The CSS is run with the classical MDS method.

The code is run at the Genomic HyperBrowser, using the web user interface. The HyperBrowser system runs on Insilico, an Abel node with 64 physical cores, further described in section 4.1.4. Since the analyses were run from the web tool, we do not have any control over the number of other jobs run on the system at the same time. The data presented are from a single run that should be representative for the performance of the system. When there are many jobs running on the HyperBrowser system, each analysis could take longer time.

5.4.1 Variable number and size of tasks

In the following, the number of threads is kept constant, while the task size is varied. The number of threads, p, is 64, and the task size is given in

$$\{10, 50, 100, 500, 1000, 5000, 10000\}$$

Cluster Separation Score

In figure 5.1, we see the runtimes for the CSS tool with a variable task size. We get the best results for a task size of 50, and for all larger task sizes, the runtime increases dramatically, suggesting that large tasks give bottlenecks for expensive regions in the chromosome.

[1] The Atlantic cod data set, se section 5.8, is more time consuming, but this data set was included late in the process.

Fisher's Exact Test

For the FET, we get the best results for a task size of 500, as can be seen in figure 5.2. This graph behaves differently than the CSS, which could be due to the fact that the windows in the FET are almost equally expensive to calculate.

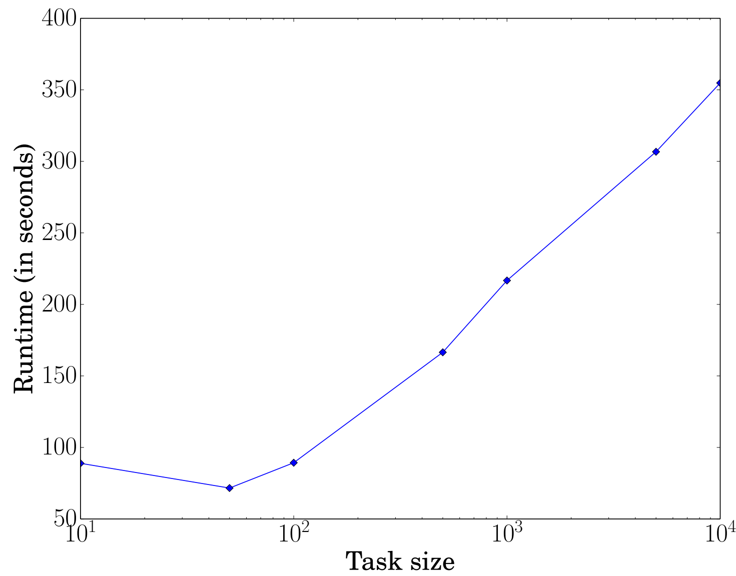


Figure 5.1: Cluster separation score (CSS) with a variable size of tasks. The number of threads is 64, and the task size is given in $\{10, 50, 100, 500, 1000, 5000, 10000\}$. The horisonal axis is plottet as a $\log_{10}$ scale.

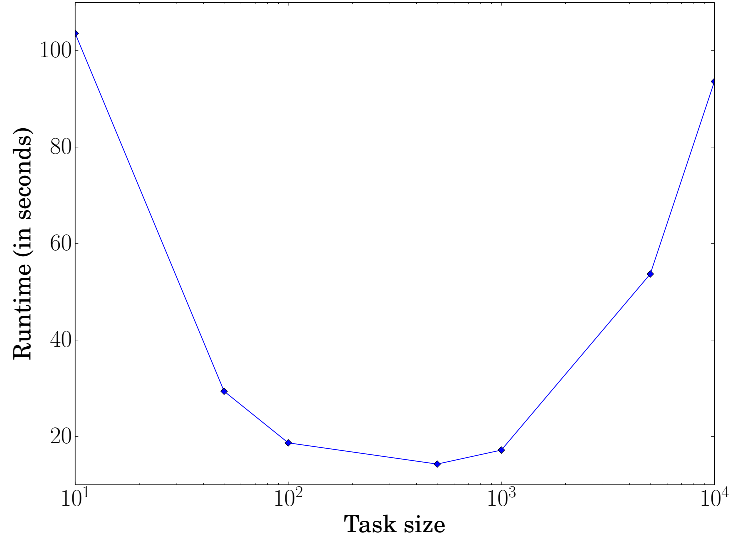


Figure 5.2: Fisher's exact test (FET) with a variable size of tasks. The number of threads is 64, and the task size is given in $\{10, 50, 100, 500, 1000, 5000, 10000\}$. The horisontal axis is plotted as a $\log_{10}$ scale.

5.4.2 Variable number of threads

In the following, the task size is kept constant, while the number of threads varies. Both the runtime and the scaled speedup are shown. The parallel program is implemented with a 'pool' of tasks, where each thread picks available tasks themselves. With the use of only one thread, this is going to be ineffective, and create a lot of overhead. To get a better picture of the speedup of the program, the scaled speedup is therefore shown. More information about the parallel implementation can be found in section 4.5.

The task size is 100 and the number of threads is given in

$$\{1, 2, 4, 8, 16, 32, 64, 128\}$$

Cluster Separation Score

As we can see in figure 5.3, the runtime for one thread is large compared to the serial version of the program, found in table 5.1. The shortest runtime is given for 128 threads, but there is not much difference between 16 to 128 threads, which indicates that the parallel program does not scale well. As we can see in figure 5.4, the scaled speedup is almost linear for the first number of threads, but it evens out for about 8 to 16 threads. We do not get much

speedup for 64 threads compared to 32 threads, which can be due to two things: overhead in the parallel program, and the fact that with a task size of 100, some regions are going to be bottlenecks. For some windows, the calculation of p-value takes a long time. If we are unlucky, some regions in the genome have several costly windows after each other. For instance, for one region of 100 windows in chromosome I, the calculation takes 14 seconds. With a task size of 100, it is impossible to lower the calculation time for this region.

To see if the size of the tasks was part of the problem, a task size of 25 was tried. In figure 5.5 we can see that for one thread, the runtime is much higher now, which is not surprising. The runtime for 64 threads is the lowest yet, with only 65.5 seconds for the entire chromosome. This indicates that at least a part of the problems in the previous run was the task size, even though overhead in the parallel program probably still plays a part.

The scaled speedup is given in figure 5.6. Here we can see that this version of the program scales better up to 64 threads, but worse for 128 threads. As hoped, we get higher scaled speedup when the task size is smaller.

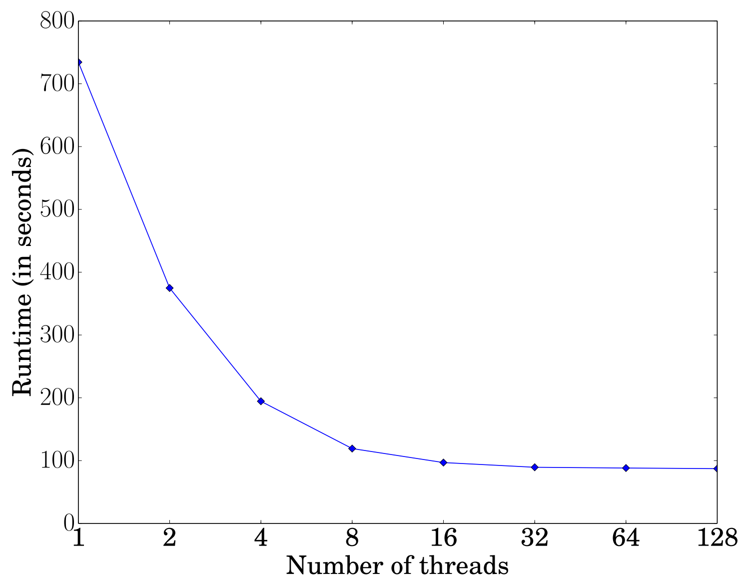


Figure 5.3: Cluster separation score (CSS) with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 100. The horisontal axis is plotted as a $\log_2$ scale.

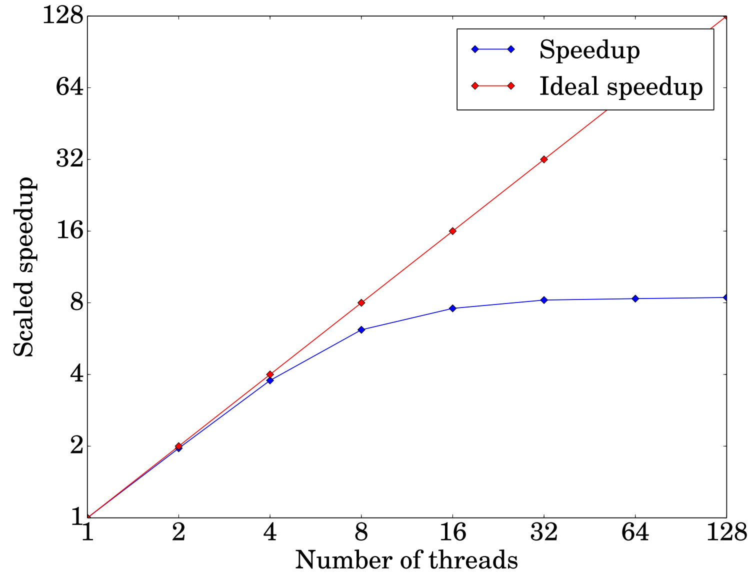


Figure 5.4: Scaled speedup (scaled for 1 thread) for cluster separation score (CSS) with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 100. The horisontal axis is plotted as a $\log_2$ scale.

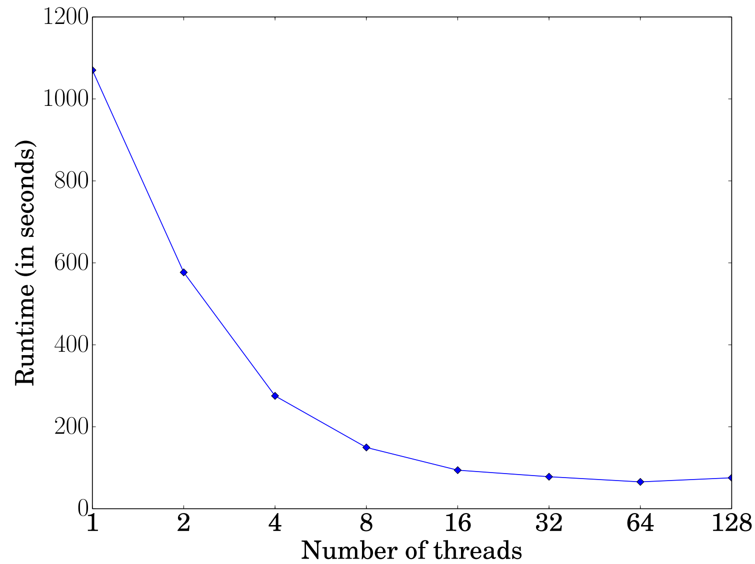


Figure 5.5: Cluster separation score (CSS) with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 25. The horisontal axis is plotted as a $\log_2$ scale.

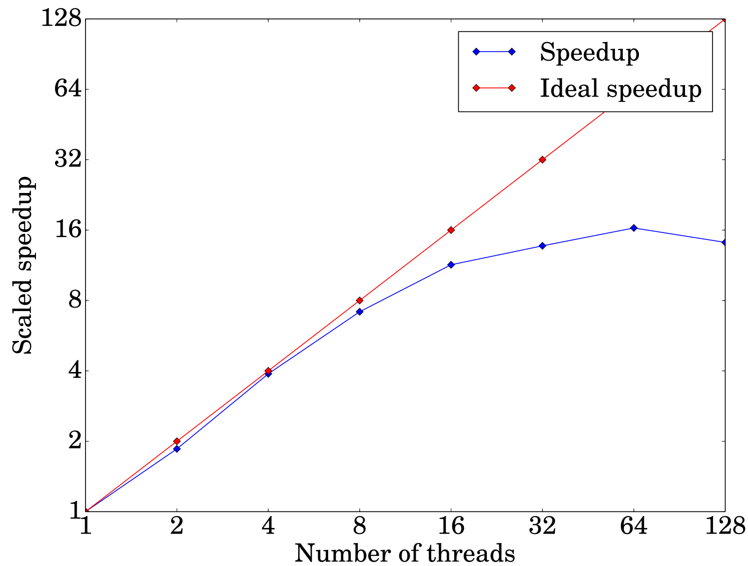


Figure 5.6: Scaled speedup (scaled for 1 thread) for cluster separation score (CSS) with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 25. The horisontal axis is plotted as a $\log_2$ scale.

Fisher's Exact Test

As we can see in figure 5.7 the shortest runtime for the FET tool is given for 64 threads. The FET scales better than the CSS, as shown in figure 5.8, and we get the best scaled speedup for $p = 64$. The speedup is close to linear for up to 8 threads, and starts to even out at 16 threads. For 128 threads, the speedup is worse than for 64 threads.

The parallel FET and CSS was implemented in the same way, suggesting that the task size was important for the results found for the CSS tool. Still, the speedup evens out for larger number of threads, suggesting that there are some overhead in the creation of threads and the fetching of tasks.

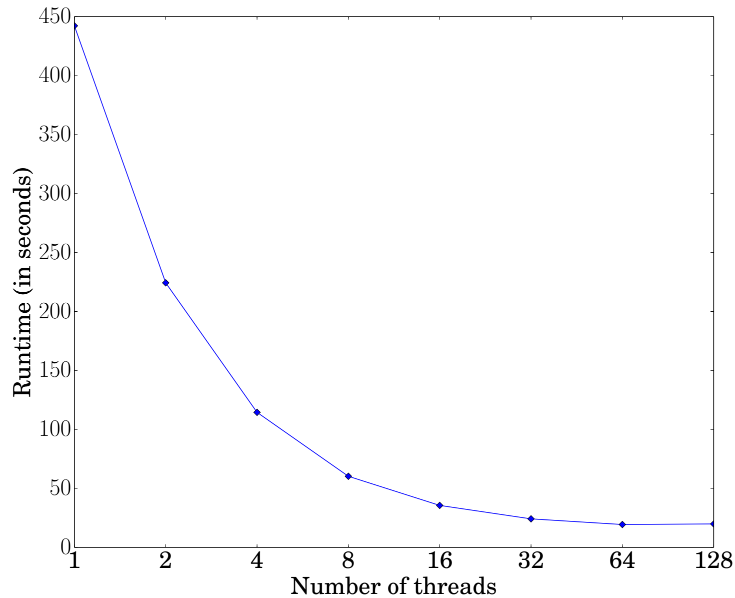


Figure 5.7: Fisher's exact test (FET) with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 100. The horisontal axis is plotted as a $\log_2$ scale.

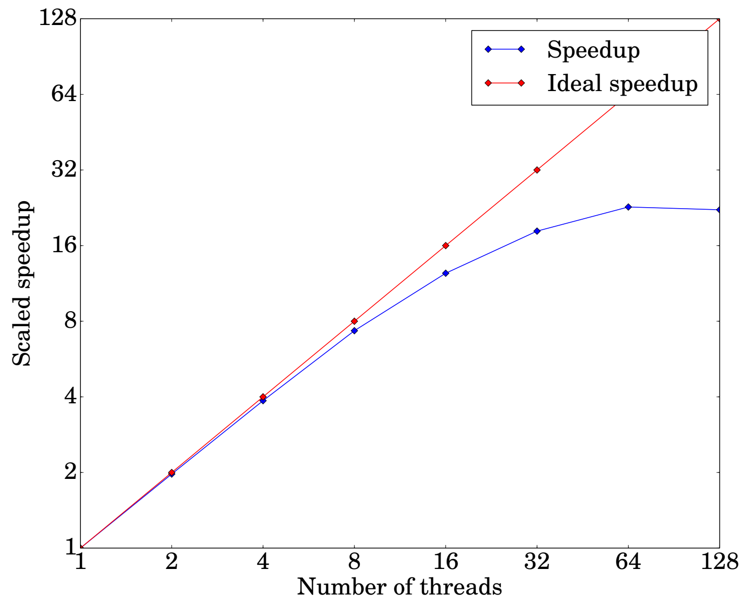


Figure 5.8: Scaled speedup (scaled for 1 thread) for Fisher's exact test (FET) with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 100. The horisontal axis is plotted as a $\log_2$ scale.

5.5 Synthetic data set

To test how well our methods work, we have created a synthetic data set with four interesting regions of different size. The goal was to see if our methods could identify these regions with various levels of noise in the data set. The data were created from the stickleback chromosome V. All the individuals (11 for population A and 10 for population B) and positions in the original data set are kept, but the SNP values are different. The SNP positions outside the four regions of interest are filled with random values in $\{3, -3, 0, -10\,000\}$. The regions of interest are filled with one SNP value, 3 (major allele) for population A and -3 (minor allele) for population B. We have four regions we want to identify, at different positions in the chrosomome:

- 1: $[100\,000, 200\,000]$, with a size of 100-kb
- 2: $[1 * 10^6, 1.2 * 10^6]$, with a size of 200-kb
- 3: $[2.7 * 10^6, 2.75 * 10^6]$, with a size of 50-kb
- 4: $[3.5 * 10^6, 3.525 * 10^6]$, with a size of 25-kb

The file has $\frac{1\,713\,085}{11} = 155\,735$ positions, distributed over the interval $[0, 12\,249\,273]$.

Noise in the data set was added by random mutations of all the values in the chromosome, with a given probability. A mutation was done by drawing a new random number from the set $\{3, -3, 0, -10\,000\}$. We have tried probabilities in $\{0, 0.2, 0.5, 0.8, 0.9, 0.95, 1\}$ and have selected some of those results for presentation. When the probability for mutation is 1 we expect a totally random data set.

5.5.1 Cluster Separation Score

All the versions of the CSS tool, with three different MDS methods, find the relevant regions for noise probabilities up to 0.8.

The analyses were done using the default values for the parameters, see secion 5.2, and the results were filtered with a false discovery rate of 0.05.

Classical MDS

Figures 5.9 to 5.12 shows the results from the Classical MDS. The method finds the correct regions for noise probabilities up to 0.8, but does not handle noise probabilities over 0.9. For noise = 0 (figure 5.9) the CMDS finds all relevant regions, without any false positives, which is only expected. For noise = 0.5 (figure 5.10) the method finds the relevant regions, but with some false positives due to noise. For noise = 0.8 (figure 5.11) we capture less false positives, but our regions are noisy and not of equal height (score value). For noise = 0.9 (figure 5.12) the method falls apart, unable to find any relevant regions. For all our noise levels in $[0, 0.8]$, the relevant regions have a $-\log_{10}(p)$-value of about 5, while for noise = 0.9, we barely get a $-\log_{10}(p)$-value above 4.

The CSS gives weaker signals for higher levels of noise. Notice that the CSS peaks in an additional region, where the $-\log_{10} p$-value does not peak. If we had filtered for the x top scoring CSS regions instead of p-value, this noise would have been included in the results.

SMACOF

The SMACOF algorithm finds the relevant regions for noise probabilites from 0 to 0.8, as we can see in figures 5.9 to 5.11. The results are similar to the results found by the CMDS, but SMACOF has slightly higher values for the CSS scores for the relevant regions, especially for higher levels of noise. For noise = 0.5 (figure 5.10), it has slightly less prominent false positives than CMDS, with a lower $-\log_{10}(p)$-value.

When the probability of noise is 0.9 (figure 5.12) and bigger, SMACOF finds no relevant regions. The results from the SMACOF algorithm is not much better than for the CMDS, with a huge increase in the runtime. As we can see, the CSS values in the noisy region is much higher for SMACOF than for CMDS.

SMACOF + CMDS

The combination algorithm (SMACOF + CMDS) finds the relevant regions for noise probabilities from 0 to 0.8, as we can see in figures 5.9 to 5.11, but finds nothing for probabilities from 0.9 (figure 5.12) and up. The combination method performs quite similar to the two other methods, but finds fewer false positives. The false positives it finds are less prominent than the false positives found by the two other methods, making it possible that a stricter limit might remove some of them completely. This method also gives a peak in the CSS values in the noisy region.

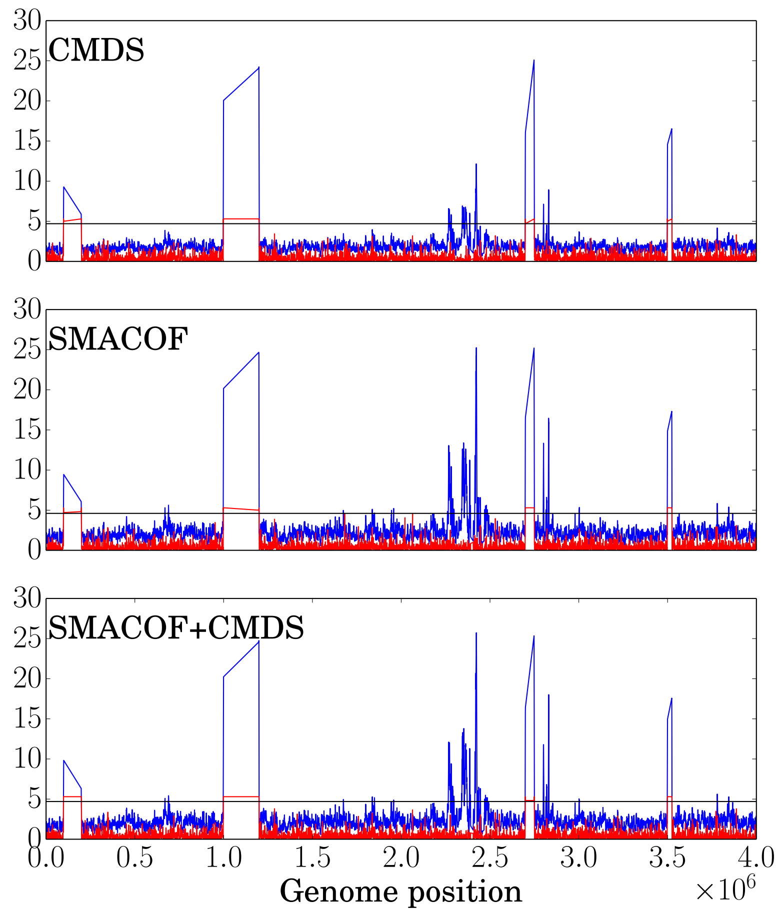


Figure 5.9: Cluster separation score (CSS) on the synthetic data, for noise = 0. The upper figure is the classical MDS, the middle is the SMACOF method and the bottom is the combination method for calculating MDS. The red lines are the $-\log_{10}(p)$ value, the blue are the CSS score, and the horisonal line is the $-\log_{10}(FDR\ p)$-value, the p-value limit for the relevant regions.

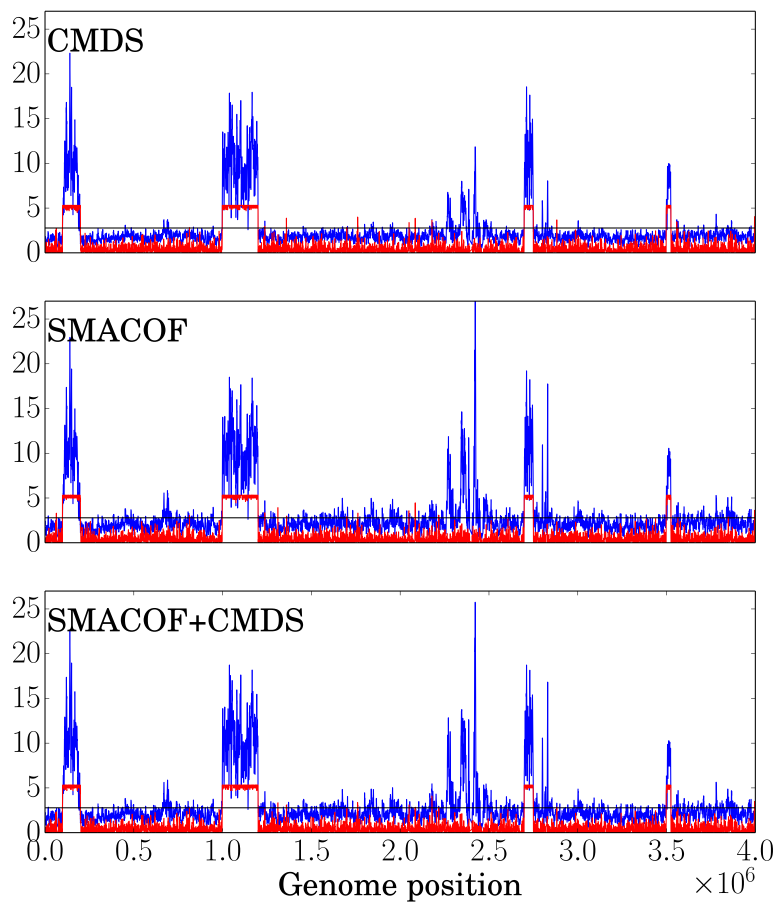


Figure 5.10: Cluster separation score (CSS) on the synthetic data, for noise = 0.5. The upper figure is the classical MDS, the middle is the SMACOF method and the bottom is the combination method for calculating MDS. The red lines are the $-\log_{10}(p)$ value, the blue are the CSS score, and the horisonal line is the $-\log_{10}(FDR\ p)$-value, the p-value limit for the relevant regions.

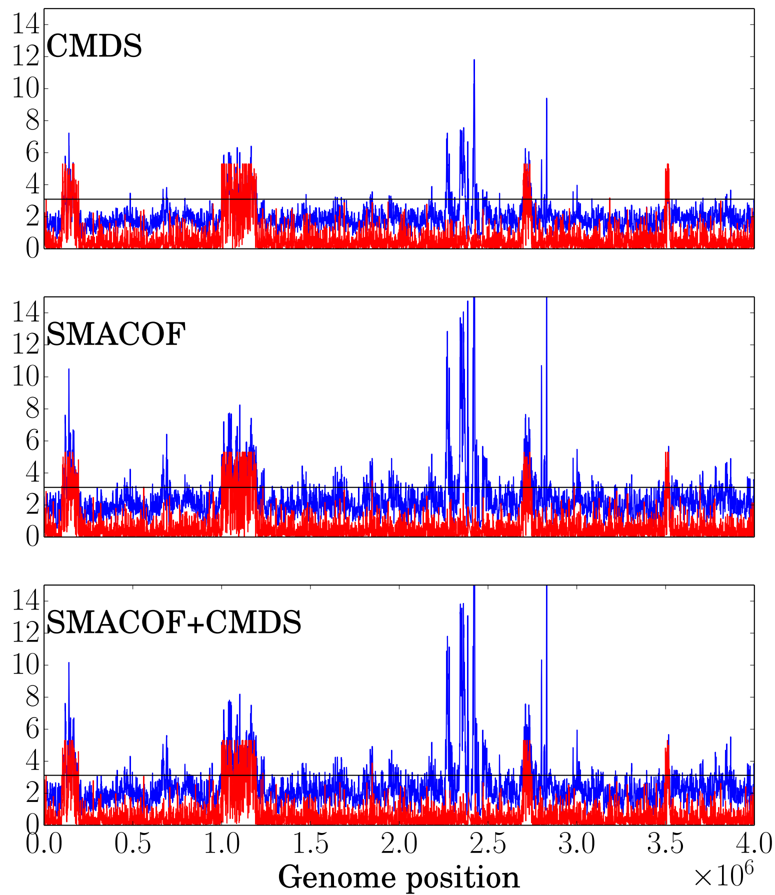


Figure 5.11: Cluster separation score (CSS) on the synthetic data, for noise = 0.8. The upper figure is the classical MDS, the middle is the SMACOF method and the bottom is the combination method for calculating MDS. The red lines are the $-\log_{10}(p)$ value, the blue are the CSS score, and the horisonal line is the $-\log_{10}(FDR\ p)$-value, the p-value limit for the relevant regions.

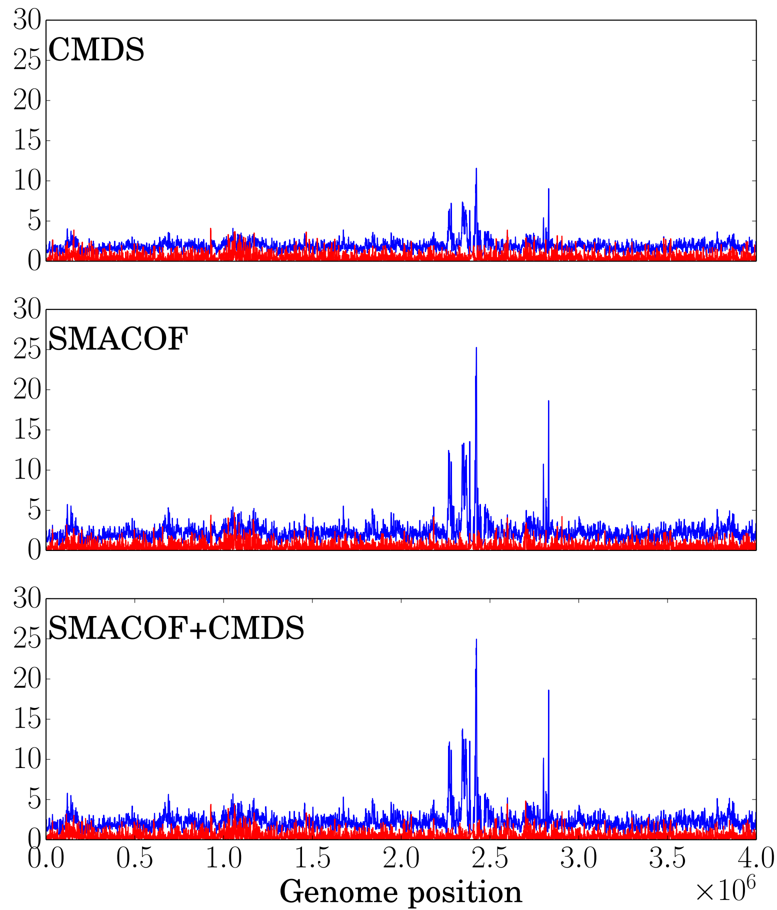


Figure 5.12: Cluster separation score (CSS) on the synthetic data, for noise = 0.9. The upper figure is the classical MDS, the middle is the SMACOF method and the bottom is the combination method for calculating MDS. The red lines are the $-\log_{10}(p)$ value and the blue are the CSS score. No relevant regions were returned, thus the lack of a p-value limit.

5.5.2 Fisher's Exact Test

In figures 5.13 to 5.17 the results for the 2.5 kb and the 100 kb FET are shown. The solid horisontal line is the limit used by Burke et al., described in section 3.2, while the dotted line is the strict limit used for the stickleback data, further described in section 5.6.2. As we can see in the figures, the FET with 2.5 kb windows gives a lot of noise in the data set. The limit used by Burke et al. does not work well on these values; for noise = 0 (figure 5.13) it captures a lot of false positives. That should not be possible. We find the relevant regions for noise $\in [0, 0.8]$ (figures 5.13 to 5.15), as for the CSS methods, but with a lot more false positives. Notice that at the position of the false region found by the CSS methods, the 2.5 kb FET gives a small peak.

For the 100 kb windows we get less noise, and the limit by Burke et al. works well. We get no false positives, which is better than for the CSS methods, and for the noise probability of 0.9 we find the relevant regions, as can be seen in figure 5.16.

For the stickleback data, the limit used by Burke et al. was too loose, and we got too many relevant regions. We therefore used a stricter limit with a 10^{-9} percentile, detailed in section 5.6.2. This limit is shown in figures 5.13 to 5.17, as the dotted line. This limit works well on the the 2.5 kb windows for noise up to 0.5, but for noise levels of 0.8 and up, it can not find any relevant regions. For the 100 kb windows, the strict limit works great for noise = 0.5, but it finds no regions when there is zero noise. This is clearly not a good limit for this data set, which indicates that the limits for the FET method must be customized for the data set, giving additional work for the user.

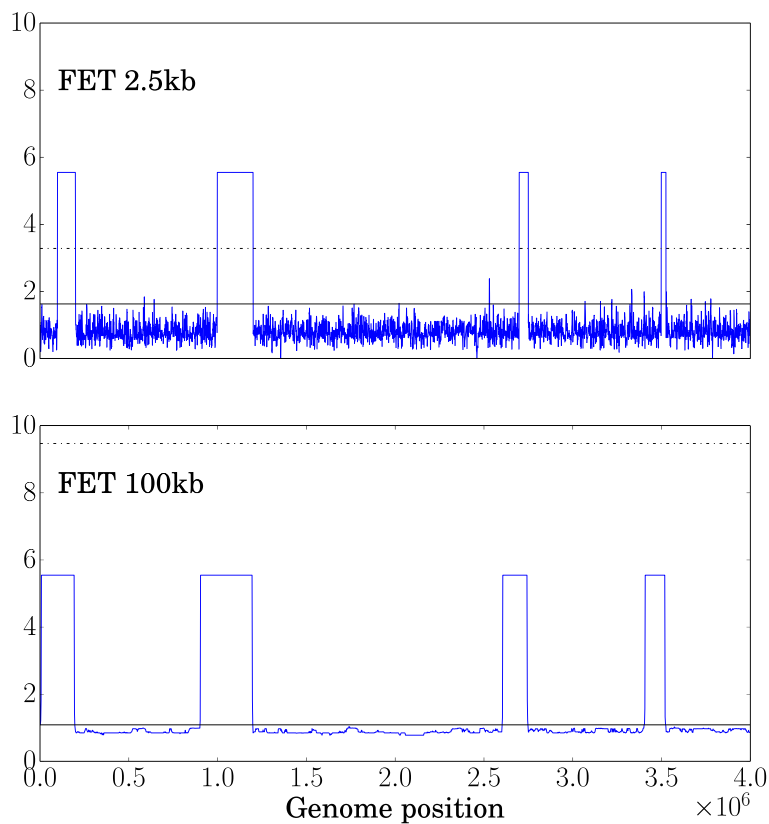


Figure 5.13: Fisher's exact test (FET) on the synthetic data, for noise = 0. The upper figure is FET with 2.5 kb windows, the bottom the 100 kb FET. There are shown two different limits. The solid line is the limit created by Burke et al. (2010), the dotted line is the limit used for the stickleback data, see section 5.6.2.

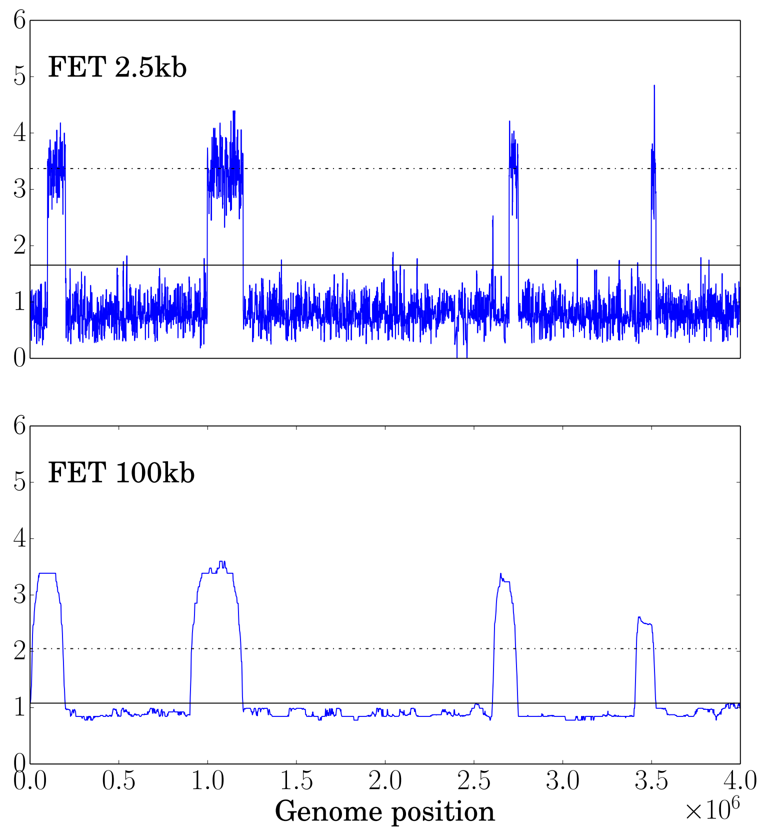


Figure 5.14: Fisher's exact test (FET) on the synthetic data, for noise = 0.5. The upper figure is FET with 2.5 kb windows, the bottom the 100 kb FET. There are shown two different limits. The solid line is the limit created by Burke et al. (2010), the dotted line is the limit used for the stickleback data, see section 5.6.2.

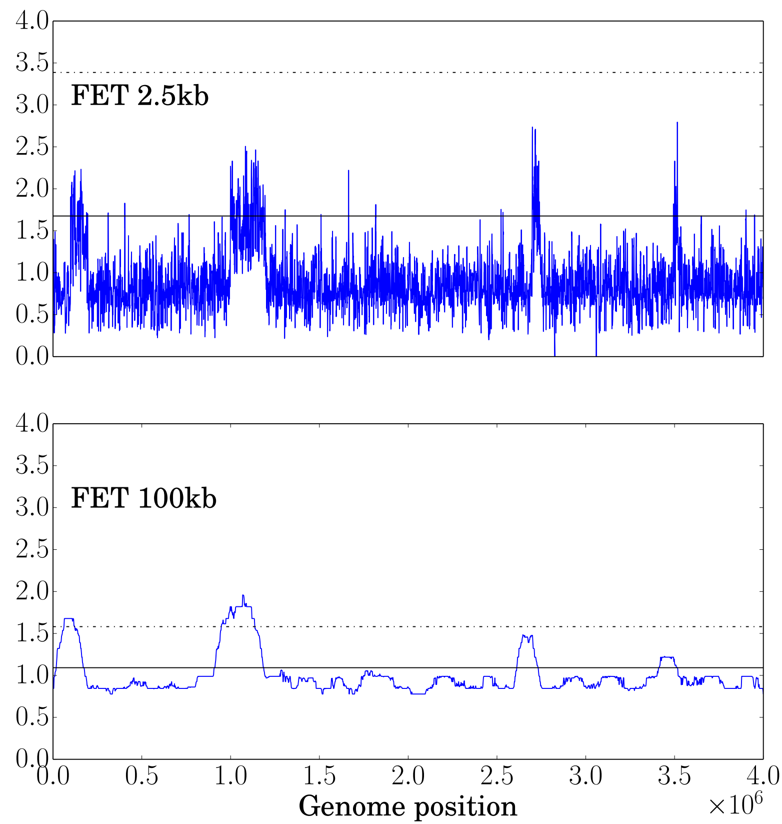


Figure 5.15: Fisher's exact test (FET) on the synthetic data, for noise = 0.8. The upper figure is FET with 2.5 kb windows, the bottom the 100 kb FET. There are shown two different limits. The solid line is the limit created by Burke et al. (2010), the dotted line is the limit used for the stickleback data, see section 5.6.2.

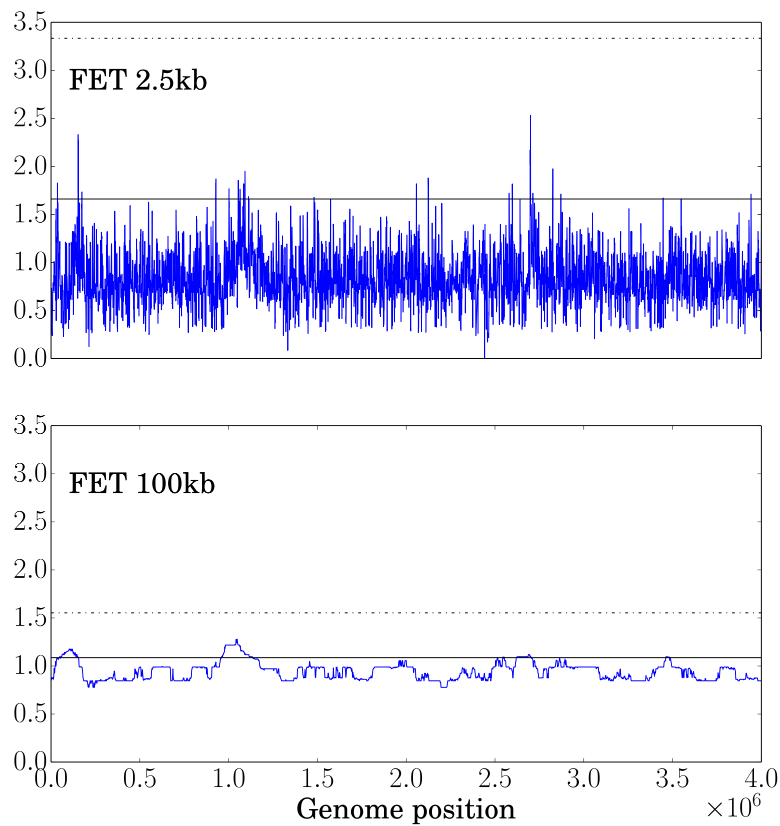


Figure 5.16: Fisher's exact test (FET) on the synthetic data, for noise = 0.9. The upper figure is FET with 2.5 kb windows, the bottom the 100 kb FET. There are shown two different limit. The solid line is the limit created by Burke et al. (2010), the dotted line is the limit used for the stickleback data, see section 5.6.2.

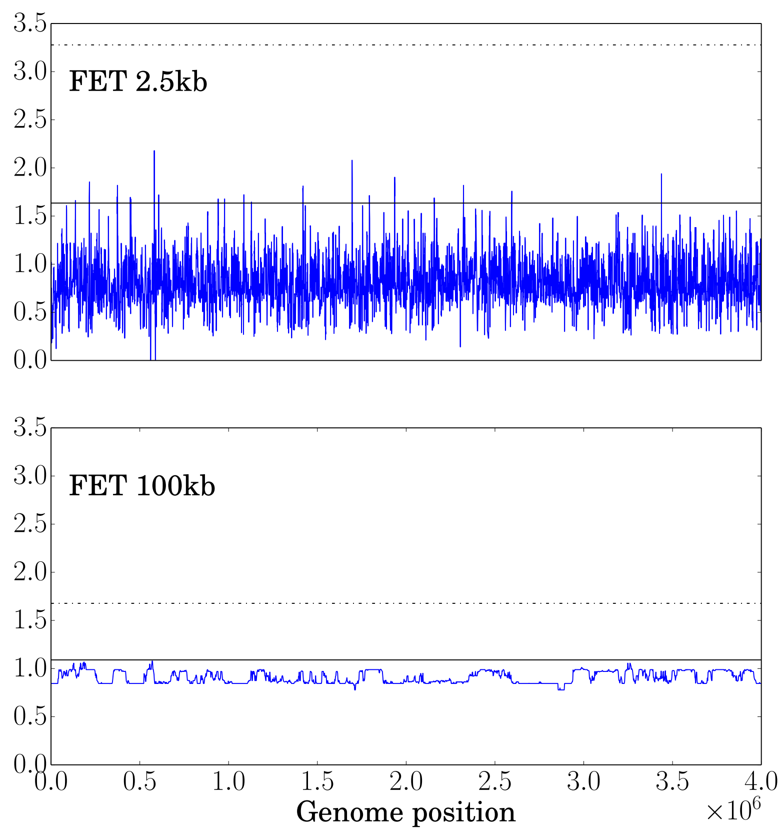


Figure 5.17: Fisher's exact test (FET) on the synthetic data, for noise = 1. The upper figure is FET with 2.5 kb windows, the bottom the 100 kb FET. There are shown two different limit. The solid line is the limit created by Burke et al. (2010), the dotted line is the limit used for the stickleback data, see section 5.6.2.

5.6 Three-spined stickleback data

The three-spined stickleback (*Gasterosteus aculeatus*) data set was obtained from the *Kingsley Lab Stickleback Genome Browser* (http://sticklebrowser.stanford.edu/). This is the data set sequenced and analysed by (Jones et al., 2012). The data set was sequenced by the Broad Institute, and the initial Genome Browser Annotations was made by Angie Hinrichs, Hiram Clawson, Kayla Smith and Donna Karolchik.

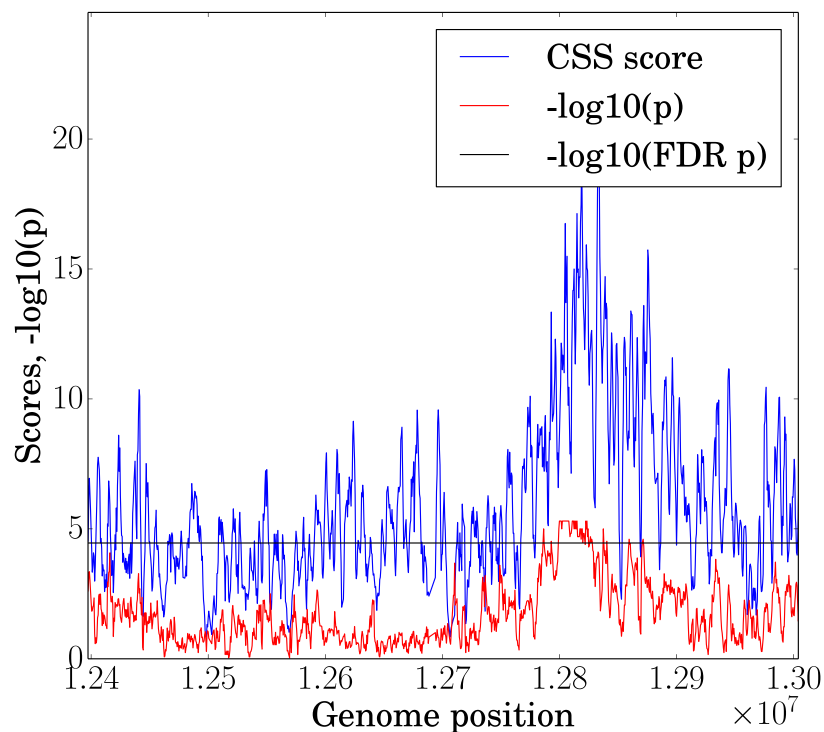
Our results are compared to the regions found by Vederhus (2013) and Jones et al. The results from Jones et al. are the intersection of their CSS method, filtered with a FDR of 0.02, and their SOM/HMM method, tree A (Jones et al., 2012). They found 55 relevant genes for marine-freshwater divergence.

5.6.1 Cluster Separation Score

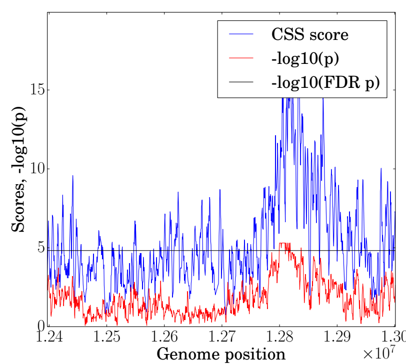
The results shown were filtered with a false discovery rate (FDR) of 0.05, and are compared to the regions found by Jones et al., and the CSS results found by Vederhus, also filtered with a FDR of 0.05.

As we can see in figure 5.18, all three methods for calculating MDS find the EDA gene. In figure 5.19 the filtered regions for chromosome VI are shown. All three methods find the same regions as Jones et al., but with more noise in the results. This is not surprising, since the regions found by Jones et al. are filtered with a FDR of 0.02, and then intersected with the results found by another method, the SOM/HMM method, which would make them more conservative. Details are given in figure 5.20, with more detailed regions in figure C.3 in the Appendix.

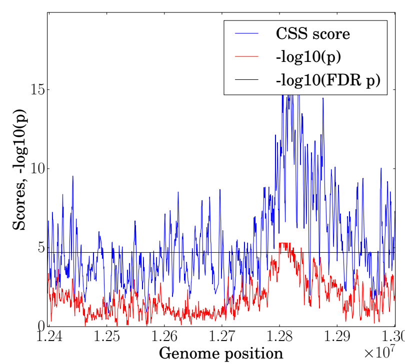
Results from chromosome XIX are shown in figure 5.21. All three methods find the same regions as Jones et al., but the CMDS finds one additional (small) region, probably noise. As can be seen in the detailed results in figure 5.22, in the relevant regions the results from CMDS matches the results found by Jones et al. best. It is interesting to see that the combination method (SMACOF + CMDS) is a middle ground between the results found by the classical MDS and the SMACOF method. Results for other stickleback chromosomes can be found in Appendix C.2.1, where we can see that the CSS finds the same regions as Jones et al., with some additional regions or noise.



(a) CMDS



(b) SMACOF



(c) SMACOF + CMDS

Figure 5.18: Results for the EDA gene in stickleback chrIV. The results shown are the cluster separation score (CSS) for all three MDS methods. The blue lines are the CSS values, the red the $-\log_{10} p$-values and the horisonal line is the FDR 0.05 limit for the p-values.

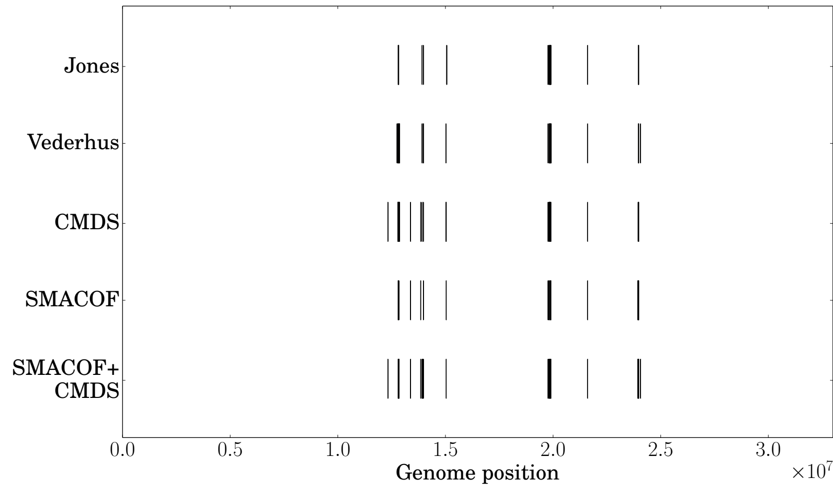


Figure 5.19: Significant regions in stickleback chrIV. The results shown are the strictest results from Jones et al., the FDR 0.05 results from Vederhus and our cluster separation score (CSS) results for all three MDS methods filtered with a FDR of 0.05.

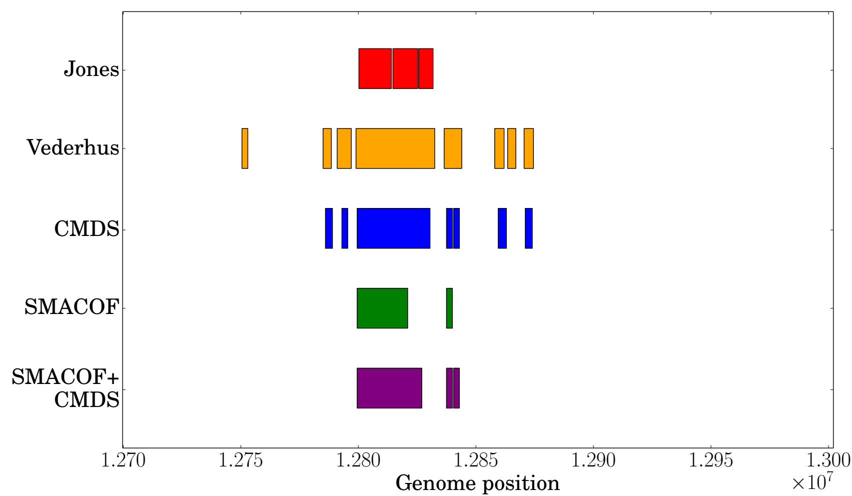


Figure 5.20: Detailed results for the EDA gene. The results shown are the strictest results from Jones et al., the FDR 0.05 results from Vederhus and our cluster separation score (CSS) results for all three MDS methods filtered with a FDR of 0.05.

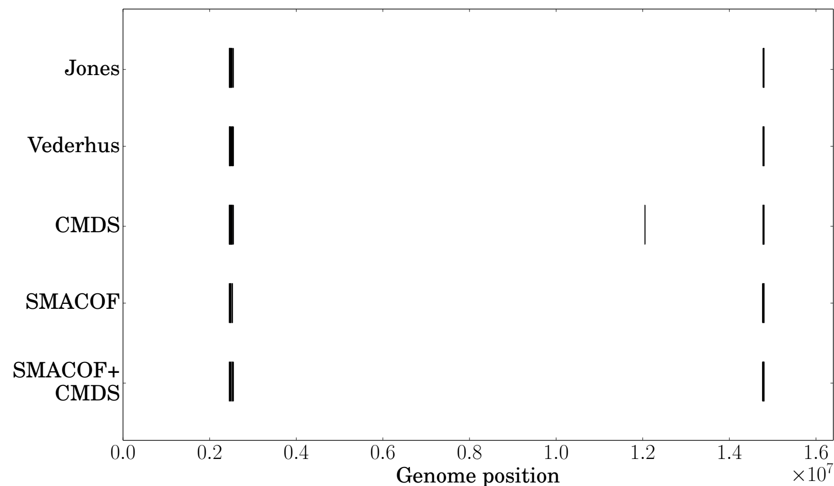
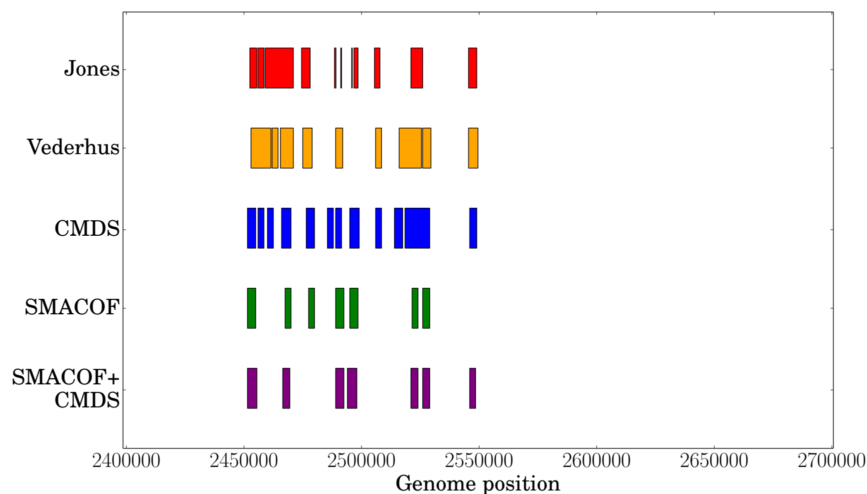
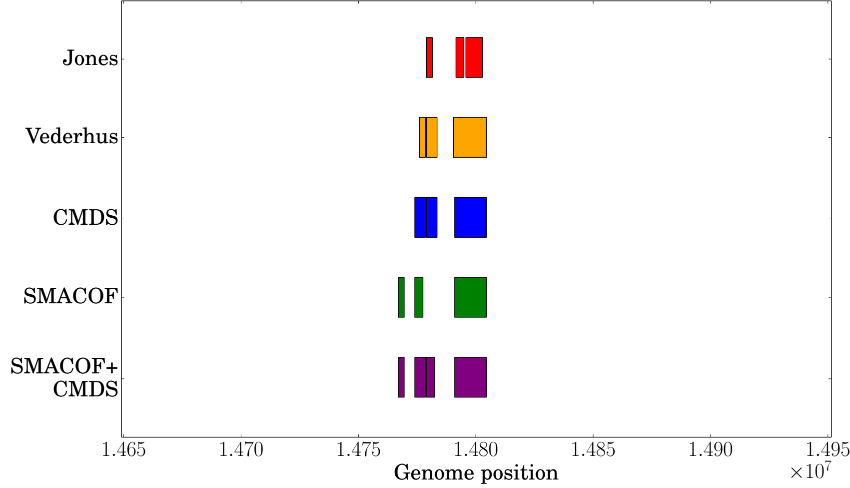


Figure 5.21: Significant regions in stickleback chrXIX. The results shown are the strictest results from Jones et al., the FDR 0.05 results from Vederhus and our cluster separation score (CSS) results for all three MDS methods filtered with a FDR of 0.05.



(a) Genome position $[2\,400\,000, 2\,700\,000]$

Figure 5.22: Detailed results for stickleback chrXIX. The results shown are the strictest results from Jones et al., the FDR 0.05 results from Vederhus and our cluster separation score (CSS) results for all three MDS methods filtered with a FDR of 0.05.



(b) Genome position $[1.465, 1.495] \times 10^7$

Figure 5.22: Detailed results for stickleback chrXIX. The results shown are the strictest results from Jones et al., the FDR 0.05 results from Vederhus and our cluster separation score (CSS) results for all three MDS methods filtered with a FDR of 0.05.

5.6.2 Fisher's Exact Test

The results shown are results from FET with two different window sizes, 2.5 kb and 100 kb. Vederhus' results are his 2.5 kb FET results.

The limit developed by Burke et al., given in section 3.2, does not work well on the stickleback data. It gives each window in the genome a probability of 0.1% to be filtered as relevant, and for the large stickleback data set, this gives too many relevant regions. This can be seen for chromosome IV in figure 5.23. As we can see, both FET methods cover almost the entire chromosome, with the 2.5 kb FET covering some additional small regions. This is clearly not good enough.

To filter fewer regions, we therefore used a different limit, developed with the help of (Vederhus, 2013) and some trial and error. For the 100 kb windows, we used the 10^{-9} percentile of the normal distribution and the 99.9th percentile of the standard deviations over all windows. This gives the following strict limit:

$$\text{median}(\text{L}_{10}\text{FET}_{5\%\text{Q}}) + \text{qnorm}(0.99999999999) \times \text{quantile}(\sigma, \text{probs} = 0.999)$$

For the 2.5kb limit, we used the 10^{-9} percentile of the normal distrbution, and the 90th percentile for the standard deviations over all windows.

Using the strict limit, the FET finds the EDA gene in chromosome IV, as can be seen in figures 5.24 and 5.26. The results for chromosome IV can be seen in figure 5.25, where we can see that both the 2.5 kb and the 100 kb FET find larger regions than Jones et al. with somewhat more noise. The FET finds more regions in the genome than the CSS, which could be due to more noise in the results. The two relevant regions in chromosome XIX are also found, as can be seen in figure 5.27.

The 100 kb FET uses larger window sizes than Jones et al. and Vederhus, and will therefore give larger regions than the other methods. The 100 FET tends to nearly miss the relevant regions found by Jones et al., by stretching out ahead of the region, and stopping to early. This is clearly visible in figures 5.24 and 5.28, where the 100 kb FET is ahead of the 2.5 kb FET.

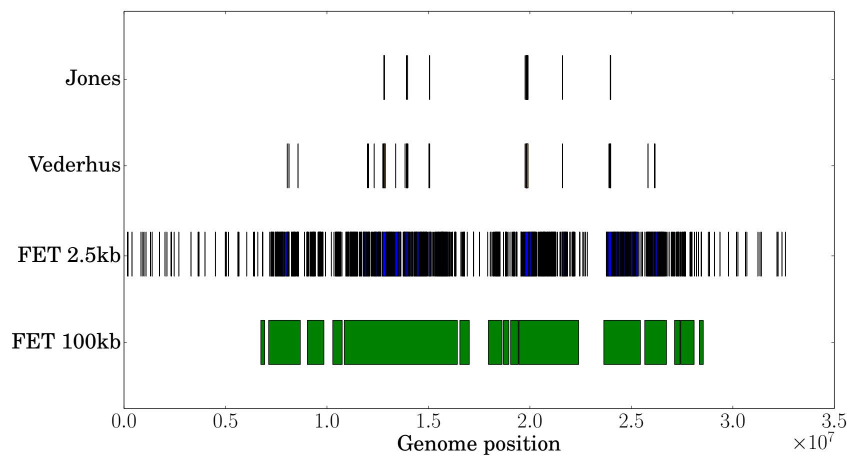


Figure 5.23: Filtered Fisher's exact test (FET) regions in stickleback chrIV. The results shown are the strictest results from Jones et al., the 2.5kb FET from Vederhus, and our FET for different window sizes, filtered with the limit given by Burke et al.

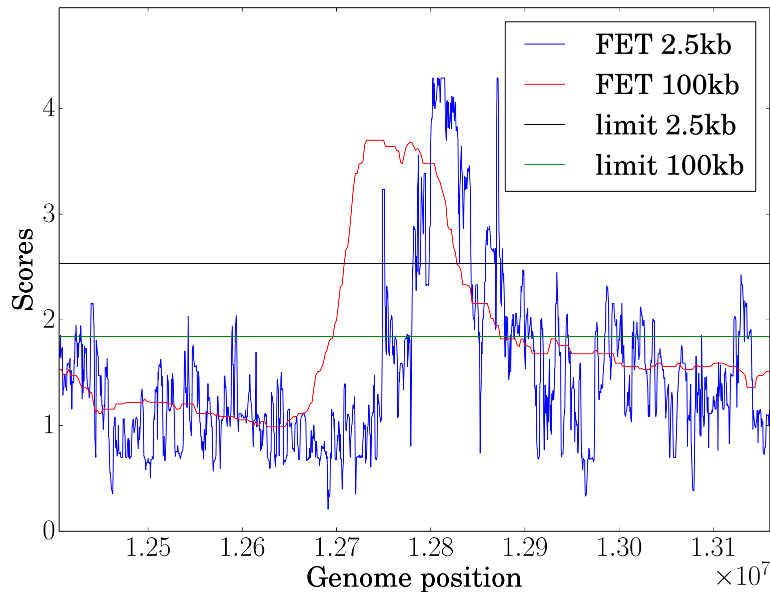


Figure 5.24: Fisher's exact test (FET) results for the EDA gene in stickleback chrIV. The results shown are the FET with windows of size 2.5 kb and 100 kb. The noisy graph is the 2.5 kb FET, the thin line the 100 kb FET. The two horisonal lines are the strict filtering limits given in section 5.6.2 for both methods. Regions above the limit are kept.

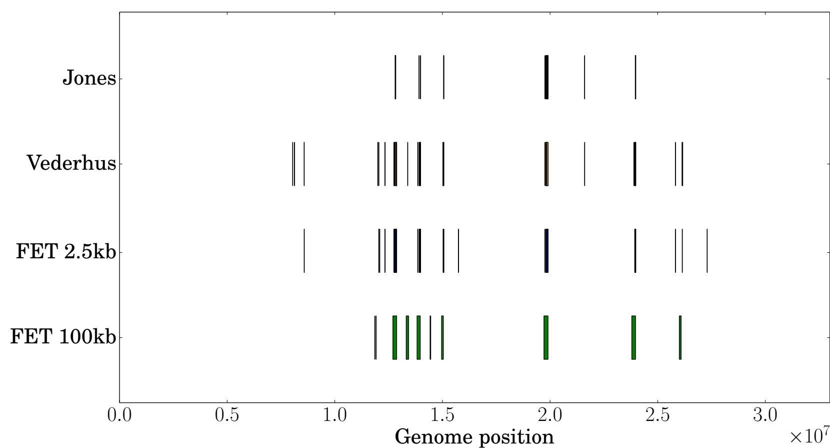


Figure 5.25: Filtered Fisher's exact test (FET) regions in stickleback chrIV. The results shown are the strictest results from Jones et al., the 2.5kb FET from Vederhus, and our FET for different window sizes, filtered with the strict limits given in section 5.6.2.

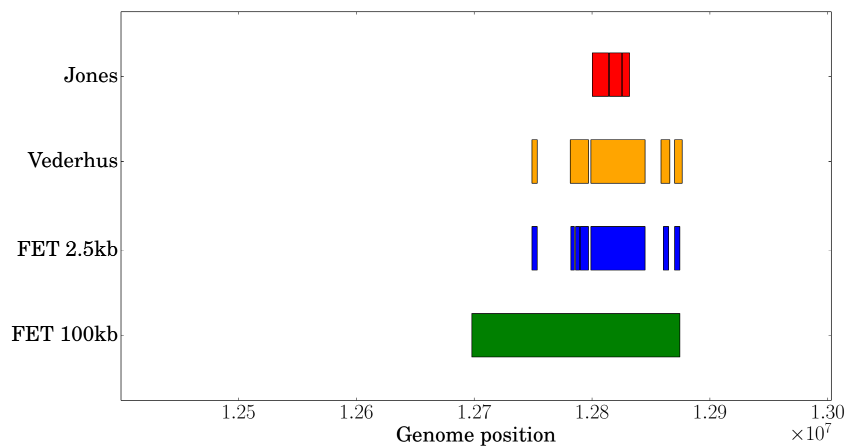


Figure 5.26: Detailed Fisher's exact test (FET) results for the EDA gene. The results shown are the strictest results from Jones et al., the 2.5kb FET from Vederhus, and our FET for different window sizes, filtered with the strict limits given in section 5.6.2.

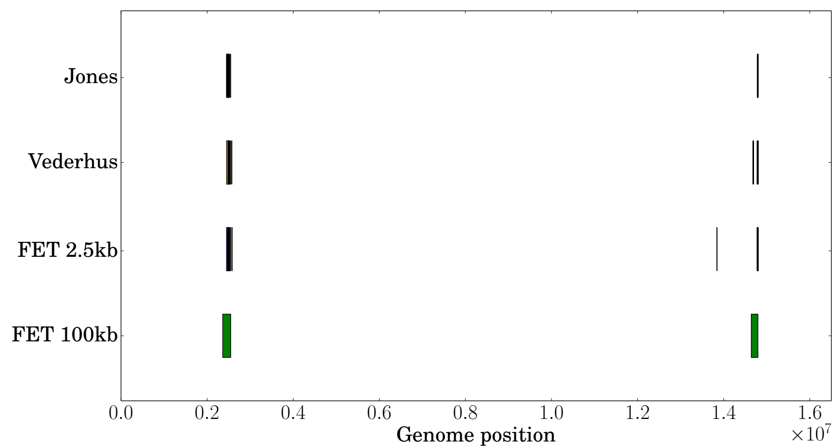
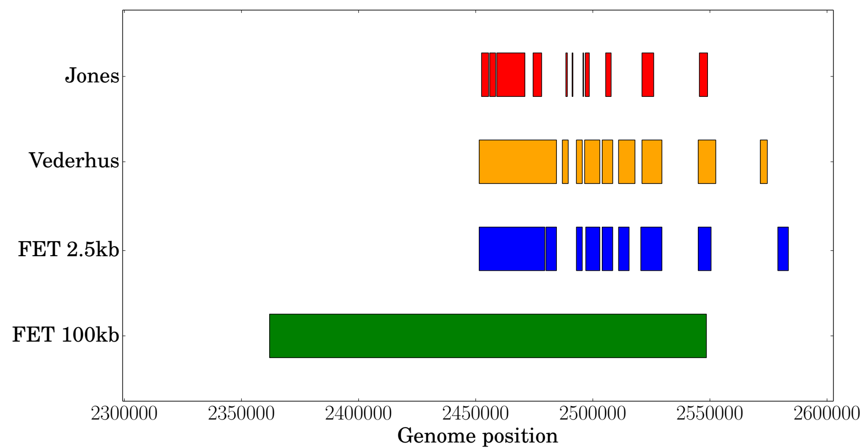
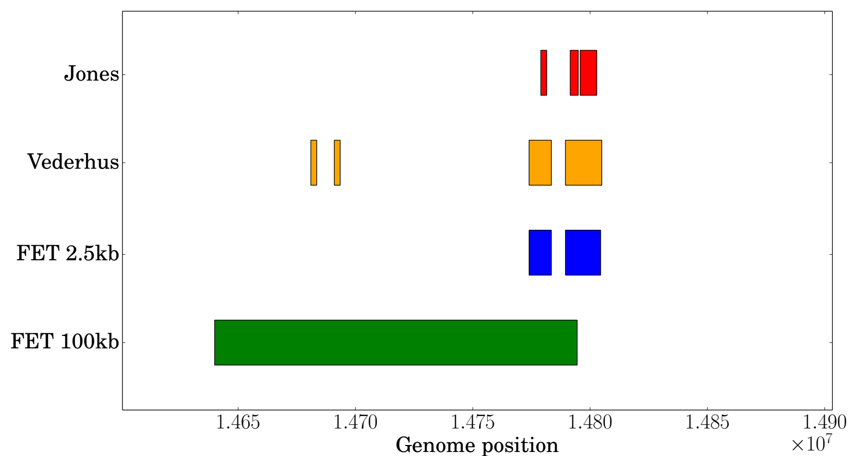


Figure 5.27: Filtered Fisher's exact test (FET) regions in stickleback chrXIX. The results shown are the strictest results from Jones et al., the 2.5kb FET from Vederhus, and our FET for different window sizes, filtered with the strict limits given in section 5.6.2.



(a) Genome position $[2\,300\,000, 2\,600\,000]$



(b) Genome position $[1.46, 1.49] \times 10^7$

Figure 5.28: Detailed Fisher's exact test (FET) results for stickleback chrXIX. The results shown are the strictest results from Jones et al., the 2.5kb FET from Vederhus, and our FET for different window sizes, filtered with the strict limit given in section 5.6.2.

5.7 *Drosophila* data

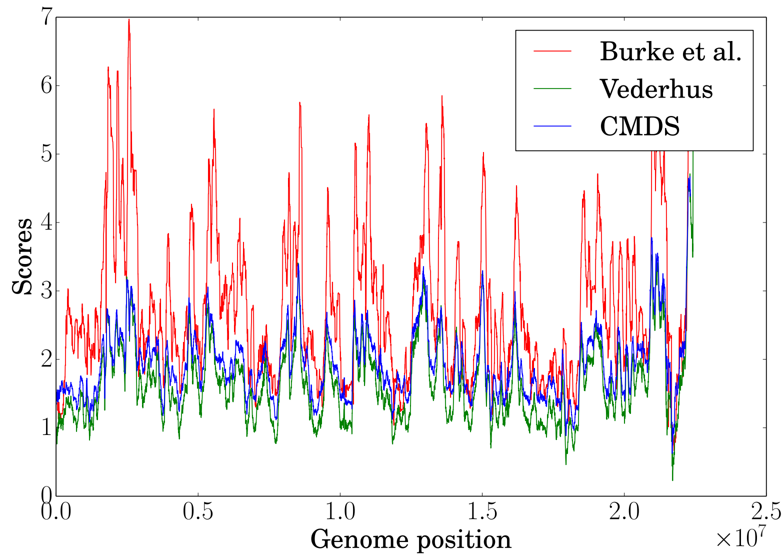
The *Drosophila melanogaster* data set was sequenced and analysed by (Burke et al., 2010), and the relevant data and results was obtained by contacting the authors. The results from our tools are compared to the results found by Vederhus (2013) and the Fisher's exact test scores from Burke et al.

5.7.1 Cluster Separation Score

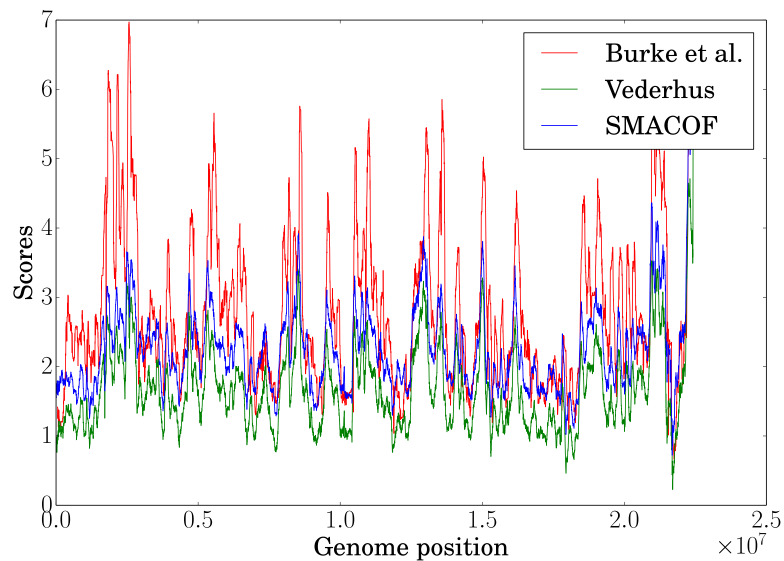
For the *Drosophila* data we only have frequency data for each population, and we can therefore only use the CSS tool, since it is not possible to create the contigency table needed by Fisher's exact test.

In figure 5.29 we see the unfiltered results for the CSS for two MDS methods, together with the results from Vederhus and Burke et al., for windows of size 100 kb. The results from Vederhus and our CSS results are multiplied by a factor of 10, to better show the similarity with the resuts from Burke et al. As we can see in figure 5.29a, the results from the CMDS fits the results from both Vederhus and Burke. Not surprisingly, it fits Vederhus' results better than those from Burke et al. For the SMACOF algorithm, the result is very similar. As we can see in figure 5.29b, the scores fit the results from Burke et al. better than the classical MDS.

Since we only have frequency values for each population, we can not generate a p-value, and will not get a proper measure of the significance of our results. We therefore need to filter the results by selecting the x top scoring CSS windows. The top 592 regions (about 1% of the scores) found for chromosome X compared with the results from Vederhus can be seen in figure 5.30. The results for all three MDS functions is quite similar, which shows that they peak at the same positions in the chromosome.



(a) Classical MDS



(b) SMACOF

Figure 5.29: Cluster separation score (CSS) results for *Drosophila* chrX. The red is the Fisher's exact test scores from Burke et al., the green is the CSS values from Vederhus and our the blue is our CSS results, for both the CMDS and SMACOF method. Our CSS results and the results from Vederhus are multiplied with a factor of 10, to better show the similarity with the results from Burke et al.

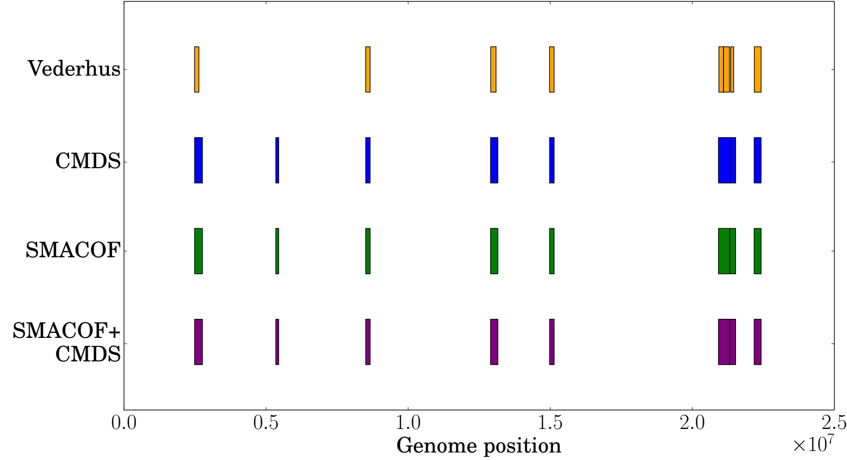


Figure 5.30: The top scoring cluster separation score (CSS) regions in *Drosophila* chrX. The results shown are the top scoring regions from Vederhus' CSS tool and our top scoring CSS regions for all three MDS methods.

5.8 Atlantic cod data

To test the tools on another data set, the Atlantic cod (*Gadus morhua*) data set was obtained from Bastiaan Star at the Centre for Ecological and Evolutionary Synthesis, at the University of Oslo. The data contains three populations of cod, one historic marine population from 1907, and two populations from 2011, one marine and one coastal. The data set contains 23 chromosomes and 76 individuals, and the data set has a mean individual genomic coverage of 7×. The data set contains over 400 000 SNPs. Divergent regions was found using the F_{st} statistic, and Star et al. (unpublished results) found 3 divergent regions between the 2011 coastal and marine populations, in chromosomes LG01, LG02 and LG07. The same regions were found between the historic population and the 2011 coastal population. For the two marine populations, there was little divergence between the regions; region 2 and 3 showed no sign of divergence, but in region 1, there was a significant difference (Star et al., unpublished results).

Two analyses were run: to compare the two marine populations, from 1907 and 2011, and to compare the coastal and marine populations from 2011. The analyses were run with a window size of 100 kb and a step size of 2 kb, since smaller window sizes were too sparse.

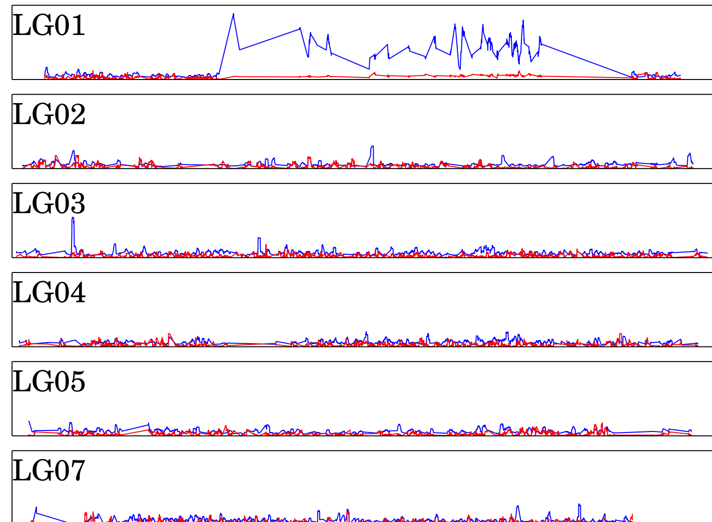


Figure 5.31: Atlantic cod, marine populations, chromosomes LG01 - LG05 and LG07: The results shown are the cluster separation score (CSS) with classical MDS with a window size of 100 kb and a step size of 2 kb. There is little divergence in the data set, thus some windows are dicarded by the CSS. This may account for some of the straight lines in the graph. The vertical axis is in $[0, 30]$, and the horisontal axis stretches through the entire chromosome.

5.8.1 The two marine populations

Cluster Separation Score

The CSS peaks in a region in LG01, similar to the region found by Star et al., as can be seen in figure 5.31. The other chromosomes show little or no divergence between the two populations, with only a few peaks in some chromosomes, the strongest in LG03 and LG17 (see Appendix C.4), probably noise.

For this data the $-\log_{10} p$-values do not peak with the CSS, as can be seen in the bottom of figure 5.32, and a filtering based on a FDR p-value of 0.05 will not identify the region. When filtering the top 500 CSS regions instead, all three MDS methods find the relevant region. This can be seen in figure 5.33.

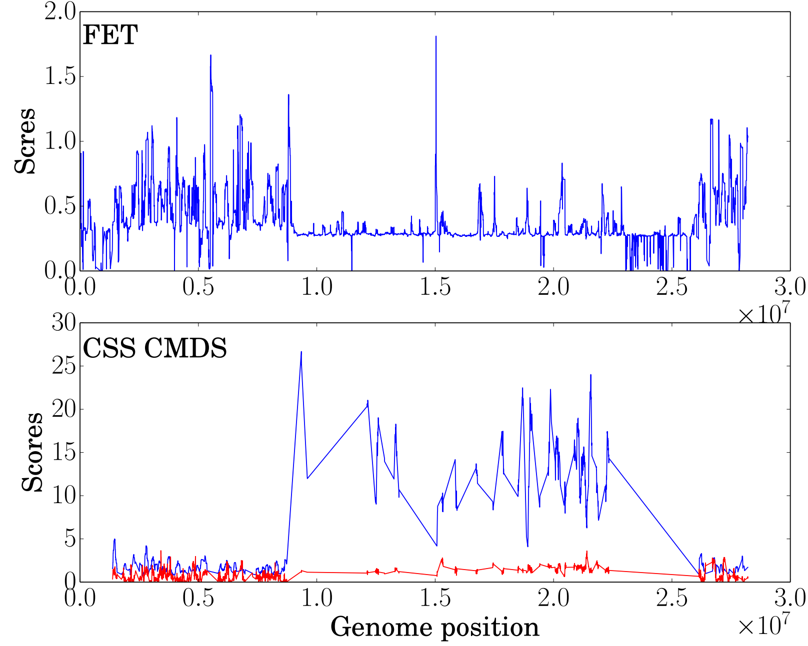


Figure 5.32: Atlantic cod, marine populations: Chromosome LG01. The top figure shows the results of a 100 kb Fisher's exact test (FET) and the bottom shows the results of the cluster separation score (CSS), run with 100 kb windows. The blue is the CSS values, the red the $-\log_{10} p$-values.

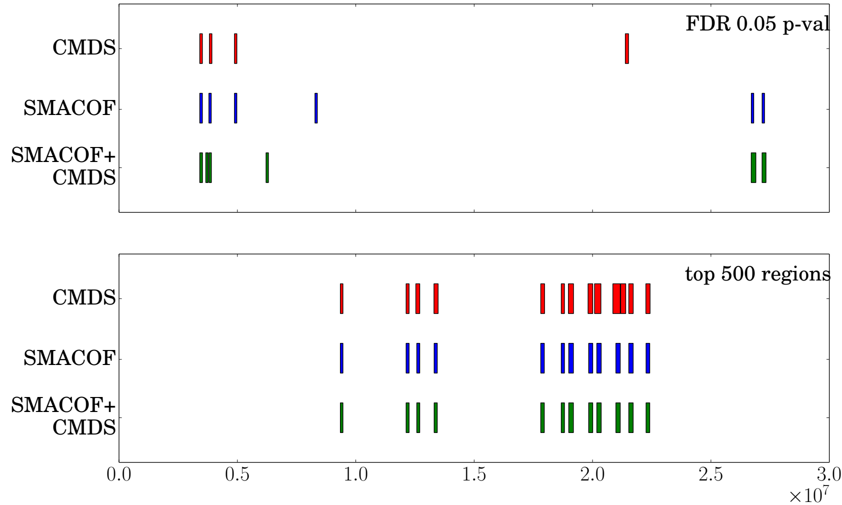


Figure 5.33: Atlantic cod, marine populations: filtered cluster separation score (CSS) regions in chromosome LG01. The top figure shows the regions filtered by a FDR p-value of 0.05, while the bottom shows the top 500 regions filtered by CSS score.

Fisher's Exact Test

The FET is not able to locate the region. The L_{10}FET score is small in the relevant region (smaller than the surrounding regions), except for one peak around position 1.5×10^7, as can be seen in figure 5.32. On this data the FET score behaves somewhat similar to the CSS p-value. Notice that the CSS has some gaps in the data, which comes from the discarded windows, while the FET will not discard any windows, and therefore has more noise in the results.

5.8.2 The marine and coastal 2011 populations

Cluster Separation Score

As we can see in figure 5.34, the CSS CMDS finds three regions of divergence, in LG01, LG02 and LG07 in what appears to be the same regions that Star et al. found, with the same length (Star et al., unpublished results). All three regions have large peaks, stretching up to 180 in LG07. The SMACOF algorithm finds less prominent peaks in the relevant regions (the largest peak is about 40), and does not find the complete region for LG07. This can be seen in figure 5.35

On this data set with more divergence, both filtering methods for CSS, the FDR p-value and the top x windows, find the relevant regions. Since there are more peaks in this data set, and fewer discarded windows, we have to select a large number of top scoring regions. This is shown for LG01 in figure 5.36, LG02 and LG07 is given in Appendix C.4.

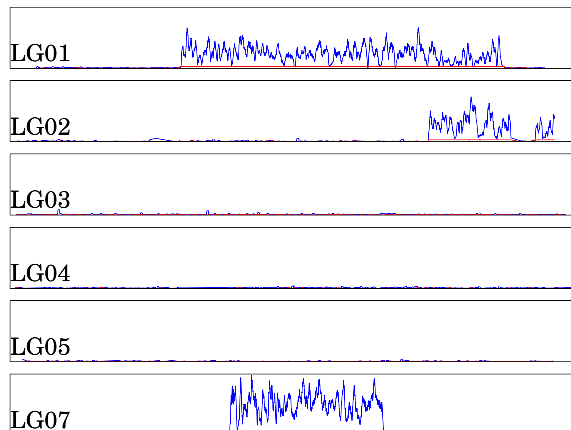


Figure 5.34: Atlantic cod, the 2011 populations, chromosomes LG01 - LG05 and LG07: The results showsn are the cluster separation score (CSS) with the classical MDS method with a window size of 100 kb and a step size of 2 kb. The vertical axis is in $[0, 180]$.

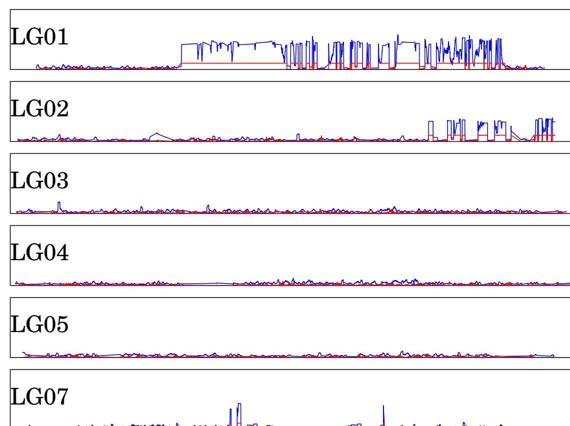


Figure 5.35: Atlantic cod, the 2011 populations, chromosomes LG01 - LG05 and LG07: The results shown are the cluster separation score (CSS) with the SMACOF method with a window size of 100 kb and a step size of 2 kb. The vertical axis is in $[0, 50]$.

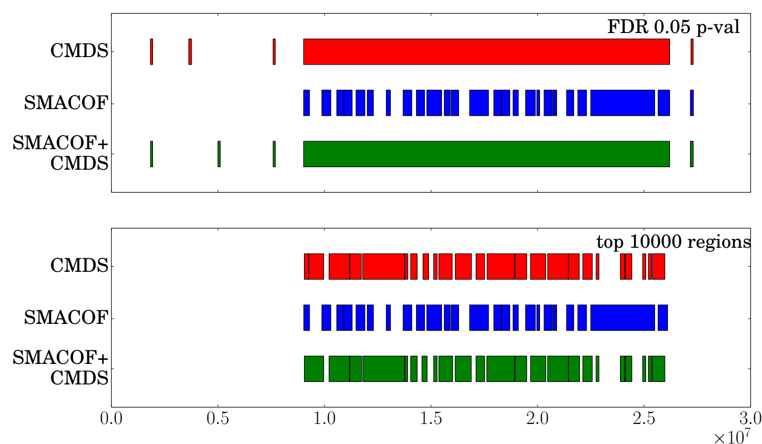


Figure 5.36: Atlantic cod, the 2011 populations: filtered cluster separation score (CSS) regions in chromosome LG01. The top figure shows the regions filtered by a FDR p-value of 0.05, while the bottom shows the top 10 000 regions filtered by CSS score. We see the results for the classical MDS, SMACOF and combination method (SMACOF+CMDS) for calculating MDS.

Fisher's Exact Test

For this data set the FET finds the three relevant regions. This can be seen in figure 5.37. When filtering out the relevant regions, the limit developed by Burke et al. (2010) returned too many regions in almost all chromosomes, so a stricter limit was used:

$$\mathrm{median}(\mathrm{L_{10}FET_{5\%Q}}) + \mathrm{qnorm}(0.999999) \times \mathrm{quantile}(\sigma, \mathrm{probs} = 0.99)$$

Chromosomes LG01 and LG09 is shown in figure 5.38, where we can see that the limit by Burke et al. gives too many relevant regions in LG09, while our limit only gives one small region here, probably noise. Both limits find the region in LG01. Chromosomes LG02 and LG07 can be found in the Appendix, figure C.21.

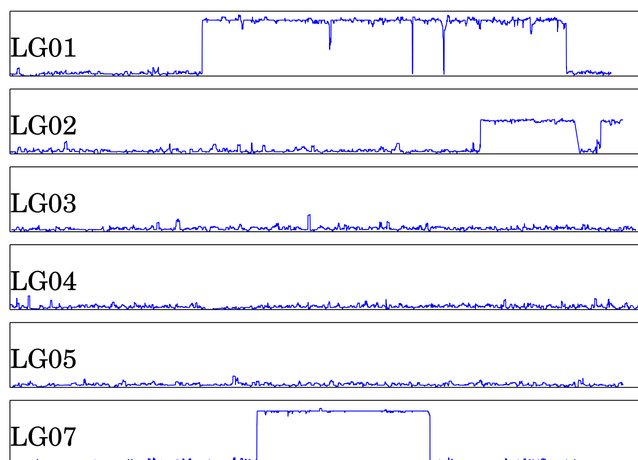


Figure 5.37: Atlantic cod, the 2011 populations, chromosomes LG01 - LG05 and LG07: Fisher's exact test (FET) with a window size of 100 kb and a step size of 2 kb. The vertical axis is in $[0, 10]$.

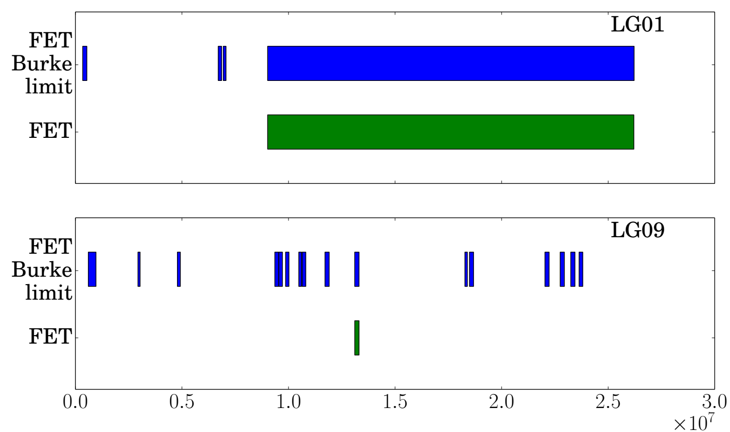


Figure 5.38: Atlantic cod, the 2011 populations: filtered Fisher's exact test (FET) regions in chromosomes LG01 and LG09 for a 100 kb FET. The results from the Burke et al. limit and the strict limit given in section 5.8.2 are shown. The top figure shows LG01 and the bottom figure shows LG09.

Chapter 6

Discussion and Conclusion

In this section the tools and results presented are discussed. The new tools have several important improvements compared to the tools made by Vederhus (2013). The most important improvements are the changes in the user interface, the increased speed of the analyses and the quality of the results. These improvements, together with possible weaknesses of the tools, are discussed in the following sections.

6.1 Discussion of the usability of the tools

The most important improvement of the user interface is the change of file format. Vederhus' old tool had to be run for each chromosome separately, creating much manual work for the user. The tool presented in this thesis can be run for the entire genome at once, giving a large improvement in the user experience. Together with the decreased run time of the tool, discussed in section 6.2, it is now feasible to run several analyses with different parameters in a short amount of time.

There are also some other changes to the user interface. The user has a larger choice of parameters, being able to vary the calculations more. Each tool is also described, with a short description or user guide at the bottom of each page. Example runs are also provided, hopefully making it easier for the users to run calculations on their own.

Even though the file format is improved, we still use the same GTrack format presented by Vederhus (2013), with two files per genome, one for each population. File formats are often problematic in bioinformatics, with many different 'standards'. To partially solve this problem, a tool for converting VCF files with SNP data to the GTrack format was created. Tools to covert other file formats to the custom GTrack is a recommended job for future applications.

We have not performed extensive user tests for the tool.

6.2 Discussion of speedup and code integration

We have achieved a large reduction of the run time of the program. Both the serial and parallel versions of the tool have a significant speedup over the tool made by Vederhus, with the serial version of the CSS tool alone achieving a speedup of $80-700\times$, depending on the MDS method used. For the stickleback data, Vederhus' CSS tool took 45 hours to run for chromosome I (Vederhus, 2013), while the new CSS tool can run for 80 - 180 seconds on the entire genome, depending on the MDS method used. The decreased runtime should make it viable for users to run several different analyses in a short amount of time.

In section 5.2 we showed that the serial C programs gained a significant speedup compared to the original Python programs made by Vederhus. This can be seen in tables 5.1 to 5.3. It is not easy to quanitfy the exact speedup of the C program versus the Python program, since the C code does more work, and/or uses different algorithms. The CSS SMACOF method, which should do the same amount of work as the Python program, achieves a speedup of $81\times$, on average, which shows that C is clearly a better choice than Python when it comes to speed.

The memory use of the program is acceptable, and is dependent on the size of the input files. It has been possible to run the C program on a normal portable computer.

The parallel program gives a considerable speedup to the serial program, making it easy to run several analyses within reasonable amount of time. Vederhus' CSS tool took 129 hours in total to run on the entire stickleback data, with a lot of manual work in between, while the new tool can be run in a couple of minutes, at worst. This makes it viable to run several analyses, with different choices of algorithms and parameters, to get good results faster.

The speedup results presented in section 5.2 and 5.4 were run on the Insilico node at the Abel computer cluster. Ideally, the analyses should be run alone on the system, with all 64 cores available for use, but since the analyses are run on the Genomic HyperBrowser from a web user interface, we do not have control of how many other jobs was run at the same time. Therefore, the maximum capacity of the system might not have been utilized.

The scaled speedup is nearly linear for up to 8 threads, but for 16 threads and up, the speedup evens out. As we can see in figure 5.8, the largest speedup is achieved for the FET method, where the tasks are of roughly equal size. Here the speedup evens out at 32 threads. This could be due to overhead of thread creation and inefficiencies in the parallel implementation. For the CSS with a task size of 100 windows, as seen in figure 5.4, the speedup is not as good as for the FET, which indicates that some of the tasks could be bottlenecks for the parallel program. For the CSS, a smaller task size gives better speedup, as we can see in the scaled speedup for a task size of 25, given in figure 5.6. The fetching of tasks are controlled by a single global variable that only one thread can access at a time. This means that smaller tasks can give increased overhead, so we must be careful and not let the task size be too small.

The integration of the C code in Python, with the use of Cython, requires

a considerable knowledge of the Python version and system hardware on the platform to use, but it is a relatively easy way to integrate the code. The Cython code is small and easy to write, and it is only the compilation phase that requires extra work. The strength of Python as a glue language is shown, and the combination of C and Python works well together. With Cython, it is easy to use different C libraries, with parallel programming being no exception.

A weakness in the runtime of the program is the time it takes to pre-process new data files the first time they are used for an analysis. With a large data set, like the stickleback data set, this takes several hours. This is a feature of the HyperBrowser system outside the scope of this thesis, but in future applications, it should be handled in some way.

6.3 Discussion of the results from analyses

In the following, the results from the analyses are discussed. The methods used are improved in some important ways. The FET method is implemented correctly; Vederhus did not calculate complete two-tailed FET, and the standard deviation was not calculated correctly. This correction should increase the quality of the results. The CSS method is implemented with three different versions of MDS: classical MDS, SMACOF and a combination of the two methods, which makes it possible to study the differences between the different methods. The user can select which method to use, and thus select the method that suits the data set best. The methods often give noise (false positives) in different parts of the genome. Therefore, it could be useful to intersect the results from the three methods, to minimize the overall noise in the results. In the following section, the results found on the synthetic data, stickleback data, *Drosophila* data and the atlantic cod data are discussed.

6.3.1 Synthetic data

All the methods were able to find the relevant regions in the synthetic data set. The CSS methods and the 2.5 kb FET could find the relevant regions for noise probabilities up to 0.8, as shown in figures 5.9 to 5.17, with some false positives in the data set. The 100 kb FET with the limit found by Burke et al. (2010) was able to find the regions for noise up to 0.9, as shown in figures 5.13 to 5.16, without any false positives. The results on this artificial data sets show that both the CSS and the FET can find relevant regions in the genome for high levels of noise. For the FET methods, two limits are shown, the limit used by Burke et al. and the limit used by us to filter the stickleback data. As we can see in figures 5.13 to 5.17, the stickleback limit does not work well on the synthetic data, especially on the 100kb FET. This is discussed further in section 6.3.2.

6.3.2 Three-spined stickleback data

On the three-spined stickleback data, the CSS method works best. The three different MDS methods give similar results, and are able to reproduce the results found by Jones et al. (2012) and Vederhus (2013) quite well. Multi-dimensional scaling is used to scale the data down to two dimensions, while keeping the distance relation between the points. The CSS then quantifies this distance between the points. Therefore it is natural that the different MDS methods do not give too different results, since they all try to preserve the distance between the same points. There are some differences between the three methods. All three find the relevant regions, but they find noise (false positives) in different parts of the chromosome, and the regions found have different length, as can be seen in figures 5.20 and 5.22. Since the different methods give noise in different areas of the genome, an intersection of the results could be used to get the best possible results.

The FET is the weakest method on this data set. The FET locates the regions found by Jones et al., for instance the EDA gene, as shown in figures 5.24 and 5.26, but the limit developed by Burke et al. does not work well on the stickleback data, returning too many regions. We needed to use a stricter limit, and ended up with a limit based on the 10^{-9} percentile of the normal distribution, as described in section 5.6.2. This limit was found with the help of (Vederhus, 2013) and some additional trial and error, making it difficult for the user to find a good enough limit without extensive study of the FET windows, for instance by plotting the results for each chromosome. The limit used for the FET should probably take into account the size of the genome, or the density of SNPs in the genome. A limit that will give each window a probability of 0.1% to be included in the top scoring regions, will naturally not work well for a large genome, with 22 chromosomes, each with tens of millions of base pairs. We will simply get too many relevant regions. In addition to this, the 100 kb FET tends to almost miss the relevant regions, with long stretches before the region, but not much after, as can be seen in figure 5.28.

Anthony D. Long, the senior author of the Burke et al. paper, has in personal correspondance said the following (Long, pers. comm.):

> *I am not so convinced the L10FET statistic we developed is the best way to go these days ... the field has progressed significantly in the last 5 years ... no-one knew what they were doing in 2010.*

This, together with our results, could indicate that the FET is not the best method available, and that the limit should be further developed.

6.3.3 *Drosophila* data

We are able to reproduce the results from Burke et al. (2010) well, see figure 5.29 and Appendix C.3. The peaks of the CSS matches the peaks produced by Vederhus and Burke et al. A problem with the result is the lack of p-values, since we only have frequency data, and no information about each individual.

The results are therefore filtered by selecting the x top scoring regions. A better limit could be developed here.

There are some possible problems with the data set. In a recent simulation study, Baldwin-Brown, Long and Thornton simulated different "evolve-and-resequence" (E&R) experiments with a different number of design parameters (Baldwin-Brown et al., 2014). They found that the parameters used by current studies did not achieve a high power of selection, and therefore suggested that E&R experiments should use larger population sizes, more generations of selection and many more independent replicates than are done by current experiments (Baldwin-Brown et al., 2014). Burke, Liti and Long empirically validated this idea in a *Saccharomyces cerevisiae* study (Burke et al., 2014), where they demonstrated that maximizing two of these parameters was necessary for identifying canditate regions in the genome. Burke et al. (2014) found a small number of highly localized regions in the genome, but when the data set was downsampled, with fewer replicate populations and fewer generations, a weaker signal involving a larger part of the genome was found (Burke et al., 2014). This indicates that the degree of replication in the experiment determines what can be learned from the data set (Burke et al., 2014).

6.3.4 Atlantic cod data

For the coastal and marine 2011 populations both the FET and CSS tool are able to reproduce the results found by Star et al. (unpublished results), as shown in figures 5.34 and 5.37. The limit used by the FET filtering method needed some work, as can be seen in figure 5.38. The CMDS method for calculating MDS worked better than the SMACOF method, with more prominent peaks. SMACOF did not find the complete region in LG07, see figure 5.35.

For the marine data set, a data set with little divergence, the FET is not able to reproduce the significant region, and gives a lot of noise in the results. The CSS is highly correlated with the F_{st} used by Star et al. (Jones et al., 2012), thus the CSS peaks in the relevant region, see figure 5.31, but the p-value does not follow the CSS, and the FDR p-value limit are not able to filter out the region.

Both the FET and the calculation of p-value used for CSS are conservative, see further discussion in section 6.3.5, which means that the tools might generate too large p-values. Thus, in data sets with little divergence, we might miss regions with small difference between the populations, and discard otherwise interesting regions. For the FET, improvements to the method can only be made in future work, but the conservativeness of the CSS p-value can be handled by the user in two ways: by studying the CSS values (and filtering on the top scoring CSS regions), or by increasing the number of runs for the Monte Carlo test. These subjects are further discussed in section 6.3.5.

The CSS discards sparse windows with too little divergence, as is discussed in section 6.3.5. For large populations with little diversity, like the two marine populations, this will lead to many discarded windows. We had to run the analyses for windows of size 100 kb, since smaller windows discarded too many

regions. The number of discarded regions can also be a problem for the p-value, as we can see in figure 5.32, where there are large gaps in the data set in the relevant region.

6.3.5 Possible weaknesses in analyses

Aside from the artificial data set, we do not have any definitive gold standard to measure our results by. The reproduction of the results from Jones et al. (2012), Burke et al. (2010) and Star et al. (unpublished results) is not enough to make a definitive conclusion, and additional data sets should be tested to evalute the performance of the methods.

There could be several sources of error: in the source code, in the parallel implementation, and in the methods themselves. Although extensive testing of each method is performed, there could be errors not found in the implementation of the methods, and race conditions not found in the parallel code. The parallel code is tested with several simple tests, to make sure it gives the same results as the serial program.

The task of this thesis was to improve the run time and user interface of the methods already implemented by Vederhus (2013). The oldest method used was presented by Burke et al. in 2010, and five years is a long time in comparative genomics. Therefore, there could be better methods for idenitifying regions of parallel divergence than using the CSS and FET methods, for instance the Nadaraya–Watson kernel regression estimate (used by R function `ksmooth`), recommended by Anthony D. Long, the senior author of the *Drosophila* paper (Long, pers. comm.). The implementation of other methods are outside the scope of this thesis, but could be done in future work.

There are several possible error sources for the CSS methods. For MDS the choice of distance metric is important, and the distance metrics used in this thesis may not be optimal. The CMDS works best with Euclidean distances, and even though it finds a best-possible solution, it minimizes Strain and not the Stress criterion used by the SMACOF method. The SMACOF method is only guaranteed to find a local optimum (Borg & Groenen, 2005), which means that we might not always get the best-possible soution. The choice of start point is important, since the starting point controls the search area of the algorithm. Two start point variants are presented in this thesis; to run the method with several random start points, as is done for the regular SMACOF method, or to select the result of the CMDS as a start point, as is done for the combination method.

The Monte Carlo test p-value, $(r + 1)/(n + 1)$, tends to overestimate the exact p-value, especially when permutations are sampled with replacement, as they are in this application. Phipson and Smyth (2010) showed that the over-estimation of the 0.05 p-value is 2% when $n = 1000$, and that small n gives a very conservative p-value. When n increases, the exact p-value converges to the estimated p-value (Phipson & Smyth, 2010). Therefore, it is necessary to run the Monte Carlo tests with many samples. To further shorten the runtime of the method, we stop the calculation when r reaches a given number. Thus, the

resulting p-value might be based on a lower sample size than n. The user can change the parameters of the Monte Carlo test as they like, and for data sets with little divergence, it is recommended to select large parameters, at the cost of increased runtime.

The CSS discards windows with too little divergence, based on the sparsity of the Δ matrix. If more than half of the values are less than 0.0001 (0), the window is discarded. For large populations with little diversity, this can lead to many discarded windows. The method for discarding windows was developed and tested on the stickleback data, a data set with a population size of 21 and relatively much diversity (Jones et al., 2012), and could therefore have some weaknesses for data sets with larger populations. The windows are discarded for a more conservative result (Jones et al., 2012), but our tool is perhaps already too conservative for data with low divergence.

For both the CSS and FET we only count the number of major and minor alleles. The positions where both alleles are observed (the '0' value in our GTrack file) are skipped, to get a more conservative result. Vederhus (2013) tried to run his analyses by counting the '0' values, but that led to much noise in the data set. This could, however, contribute to the overall conservativeness of our results.

The statistical power of FET has generated heated debate in statistical literature, the main criticism being that the test is too conservative (i.e. that it generates too large p-values), which can increase the resources needed for clinical trials (Crans & Shuster, 2008). Crans and Shuster found that the FET was too conservative even for sample sizes up to 125 per group, and proposed adjustments to the FET which would increase the statistical power, by making it less conservative. The sample sizes of the stickleback data are 10 and 11 per group, which means that the results found could be too conservative, and that greater population sizes are needed for better results.

6.4 Weakness in implementation

A weakness in the implementation is that some changes had to be made to the current HyperBrowser system to make these tools work. We have e.g. changed the sorting algorithm used in the pre-processing of input files. The CSS tool assumes that the files are ordered by position and then individual, in the same order throughout the data. With a different sorting algorithm that does not preserve the correct order of the elements, for instance quick sort, the CSS test will not give correct results. This should be handled before the code is merged into the standard HyperBrowser code.

6.5 Conclusion

In this study we improved the existing tools for comparative genomics, by improving the file format and user interface, and achieved a large speedup of the

tools. The tools are now much more practical to use and the user should be able to run several analyses with different parameters in a short time. A tool for converting VCF files to the GTrack format is also developed, to make the tools usable for a broader range of applications. We have decreased the runtime of the tools dramatically, which should make the tools possible to use in future studies, with the possibility of larger data sets.

Our tools are able to reproduce the results found by previous studies. The CSS gives accurate results on a broad range of data sets. The three different MDS methods often give similar results, but with noise in different parts of the genome. On some data sets, the CMDS gives the strongest signal, while on other data sets the intersection of the tree MDS methods will give the best result.

The FET works well on several data sets, but is weaker than the CSS. The limit used to filter the FET scores does not work well on all data sets, and needs a lot of trial and error by the user. This limit therefore needs to be developed further. For data sets with overall low divergence the FET value and CSS p-value might be too conservative, rejecting regions with small, but possibly important, divergence. The p-value calculation can be improved by the user, by selecting larger parameters for the Monte Carlo test, at the cost of increased runtime.

The increased speed, usability and accuracy of the tools make it feasible to run these analyses on a larger scale than has previously been done. These tools should therefore be able to meet the increased need for analysing large scale data sets in comparative genomics.

Chapter 7

Future Work

We have showed that we were able to improve the user interface of the tools, and have achieved a large speedup of the tools. We are also able to reproduce the results found by previous work, but there are still some improvements that could be made to the tools in future work.

To further increase the runtime of the tools, the calculations could be ported to Abel, the computer cluster at the University of Oslo. The current parallelization is done on Insilico, using threads, with parallelization inside each chromosome. It should therefore be relatively easy to port the calculations on to Abel. This could achieve a larger speedup of the calculations. The different methods used could also be implemented using faster algorithms. There are for instance other dimensionality reduction algorithms that could be tried, for instance Split-and-Combine MDS (Tzeng, Lu & Li, 2008), Isomap (Tenenbaum, de Silva & Langford, 2000) or Local Linear Embedding (Roweis & Saul, 2000).

The methods used in these tools are perhaps no longer the 'state of the art' in bioinformatics, and the tools should be extended with several new methods, for instance with the Nadaraya–Watson kernel regression estimate (used by R function `ksmooth`), recommended by Anthony D. Long, the senior author of the *Drosophila* paper (Long, pers. comm.). The calculation of limit in the FET tool should also be developed further. Vederhus (2013) suggested that a limit based on FDR could be useful.

For data sets with little genomic divergence, the FET and the CSS p-value calulated might be too conservative. The FET could be made less conservative by the suggestions made by Crans and Shuster (2008). Phipson and Smyth (2010) have several suggestions for a different calculation of the Monte Carlo test p-value, such as calculating the p-value without replacement, or calculating the exact p-value, which could be implemented in future works.

There are some improvements that could be made for the usability of the tools. One recommended feature would be to automatically run analyses with several different window sizes set by the user. The file format used is a custiom GTrack file, and even though we have created a tool for converting VCF files to the custom GTrack format, tools for converting other file formats to our custom

GTrack format could be developed.

The output from the tools must now be analysed and plotted by the user, and this requires knowledge of a good plotting tool or some programming skills. A future improvement could be to let the tools generate graphs and barplots from the results, for each chromosome or for chromosomes selected by the user.

The order of the input data to the CSS tool is important. Therefore we have changed a part of the HyperBrowser code, as described in section 4.9. This must be handled if the comparative branch is going to be merged in to the regular HyperBrowser branch. Either, a choice of sorting algorithm could be implemented, so that the different tools could choose which sorting algorithm to use, or the CSS tool must be modified to handle input data with random order of individuals.

Appendices

Appendix A

Example runs

Both example runs presented in this chapter can be found at
https://hyperbrowser.uio.no/comparative/u/tuvakt/p/example-runs.

A.1 An example run with three-spined stickleback data

A.1.1 Converting the three-spined stickleback data file

The data files from (Jones et al., 2012) are available on the Sticklebrowser, http://sticklebrowser.stanford.edu/. The relevant data files can be obtained from the Table Browser (Karolchik et al., 2004). Go to http://sticklebrowser.stanford.edu/cgi-bin/hgTables and download the 'Visual Genotype' track for the entire genome. This file is big, so it should be downladed as a gzip file. The relevant choices for the Table Browser can be found in figure A.1. On the Table Browser, the results from (Jones et al., 2012) can also be found.

The data must be loaded into the Genomic HyperBrowser with the 'Upload File' tool. If the file is big, it can be wise to load it as a compressed file, e.g. a .zip file. The upload will be faster, and the HyperBrowser unpacks the data. It is important not to refresh the browser while uploading a file, this can spoil the upload.

The data must be converted to the correct GTrack format. This is done with the comparative genomics tool **Convert Stickleback Snps to Gtrack**. This can be seen in figure A.2. The conversion needs to be done twice, once for each population group. If the files are big, as in this case, this tool will take some time to run. This tool is one of Vederhus' old tools. We have only made a small change to the header of the GTrack file.

When the conversion is finished, the FET and/or the CSS tool can be run on the data.

Table Browser

Use this program to retrieve the data associated with a track in text format, to calculate intersections between tracks, and to retrieve DNA sequence covered by a track. For help in using this application see Using the Table Browser for a description of the controls in this form, the User's Guide for general information and sample queries, and the OpenHelix Table Browser tutorial for a narrated presentation of the software features and usage. For more complex queries, you may want to use Galaxy or our public MySQL server. To examine the biological function of your set through annotation enrichments, send the data to GREAT. Refer to the Credits page for the list of contributors and usage restrictions associated with these data. All tables can be downloaded in their entirety from the Sequence and Annotation Downloads page.

clade: Vertebrate **genome:** Stickleback **assembly:** Feb. 2006 (Broad/gasAcu1)

group: Jones et al 2012 **track:** Visual Genotype

add custom tracks track hubs

table: visualgenotype describe table schema

region: genome position chrIV:12750000-13000000 lookup define regions

identifiers (names/accessions): paste list upload list

filter: create

intersection: create

output format: all fields from selected table Send output to Galaxy GREAT

output file: (leave blank to keep output in browser)

file type returned: plain text gzip compressed

get output summary/statistics

To reset **all** user cart settings (including custom tracks), click here.

Figure A.1: The Table Browser (http://sticklebrowser.stanford.edu/cgi-bin/hgTables). Here the files and results from Jones et al. (2012) can be found. Pictured are the relevant choices for downloading the stickleback SNP data from Jones et al.

Convert from SNPs in special 'Jones et al' format to gtrack files

SNP file 1 - Jones

Comma-separated list over IDs to extract

0,1,2,3,4,5,6,7,8,9,10

Execute

Corresponding batch command line:

Figure A.2: The Convert Stickleback Snps to Gtrack tool web user interface https://hyperbrowser.uio.no/comparative. This tools converts the SNP file from Jones et al. (2012) to the GTrack file format.

A.1.2 The Fisher's Exact Test Tool

Figure A.3: Fisher Exact Test SNP Tool web user interface, https://hyperbrowser.uio.no/comparative. This tool calculates a FET score and a standard deviation for each window in the genome. The user can select the window and step size, and the percentile FET score to represents the window. The default values are 2.5 kb, 0.5 kb and 0.95, respectively.

To run the Fisher's exact test, choose the **Fisher Exact Test SNP Tool** under comparative genomics. There are several choices. The user interface can be seen in figure A.3.

- First, the correct genome build must be selected. For the stickleback data, this is 'Stickleback Feb. 2006'.
- The next step is to select the SNP data for the two groups. This is the two converted GTrack files from history.
- The sliding window size and step must be selected. The default here is 2500 and 500.
- The percentile of the FET scores for each window must be selected. The default is 0.95, which corresponds to the 95th percentile of the scores in each window.

- There are two different output formats. 'tabular' gives an output file with the results (needed for further analysis), while 'html' is good for debugging/running.

Press 'Execute', and the analysis is started. First time one of the tools is run on a new file, the file is pre-processed. This is part of the standard HyperBrowser code and is run for each analysis. If the file is big, this takes a long time, e.g. for the stickleback data this can take several hours. The file is only pre-processed the first time it is run. Next time the data set is used it will run as normal. The output from the FET tool (for the 'tabular' output format) is a text file with all the results, ordered by chromosome and the start position of the window. The file looks like this:

```
#seqid  start   score           stddev
chrI    0       0.716003343635  0.249703127287
chrI    500     0.689210167047  0.195583453385
chrI    1000    0.689210167047  0.208238581515
```

To filter out the relevant regions, use the **Filter Fisher Scores** tool, also under comparative genomics. The input to this tool is the resulting file from the FET tool. The web user interface of the tool can be seen in figure A.4. Here the relevant FET file and genome build must be selected, and the user have the choice of some additional parameters. The output file is a GTrack file with the relevant regions in the genome.

A.1.3 The Cluster Separation Score Tool

To run the CSS analysis, the **Cluster Separation Score** tool must be selected. The user interface can be seen in figure A.6. Many of the choices here are similar too the FET tool, with some exceptions:

- A comparison metric must be selected. For the stickleback data, select 'Count individual SNP-differences in window'. For frequency data, select 'Compute average of difference ...'
- The parameters for the Monte Carlo test must be selected, the 'Minimun significance score runs' and 'Maximal significance scores'. The first controls the minumim amount of 'hits' needed, the other controls the maximum number of runs. Larger values gives more accurate results, at the cost of increased runtime.

To filter the results found by the CSS the tool **Significant CSS Regions** is used. The user interface of this tool is shown in figure A.5. The user has two filtering options: To filter the p-value by a FDR rate, or to select the x top scoring CSS regions. The FDR filtering scheme is recommended, but it can only be used on data with information of each individual, as for this stickleback data set, and not pooled populations, like the *Drosophila* data (Burke et al., 2010).

Figure A.4: Filter Fisher Scores web user interface https://hyperbrowser.uio.no/comparative. This tool filters the windows found by the FET SNP Tool, to return the most prominent regions. The user selects the parameters of the filtering.

Figure A.5: Significant CSS Regions web user interface https://hyperbrowser.uio.no/comparative. With this tool, the windows found with the CSS tool can be filtered with two different methods: by a FDR p-value limit, or select the top x CSS regions. The filtering method is selected by the user.

Calculate cluster separation score :)

Genome build: Stickleback Feb. 2006

Group 1 SNPs 2 - Convert Stickleback Snps To Gtrack

Group 2 SNPs 3 - Convert Stickleback Snps To Gtrack

Pairwise individual comparison metric Count individual SNP-differences in window

Sliding Window Size

2500

Sliding Window Step

500

Minimum significance score runs

10

Maximal significance scores

200000

Output format tabular

Execute

Corresponding batch command line:

Figure A.6: Cluster Separation Score web user interface https://hyperbrowser.uio.no/comparative. This tool calculates a cluster separation score and the corresponding p-value for each window in the genome. The user selects the window and step sizes. The tool estimates the significance by a Monte Carlo test. Parameters for this analyses must also be selected.

A.2 VCF Convert example run

This tool converts a VCF file with SNP data to our custom GTrack format. We made some assumptions about this file, these can be found in section 4.2.2. The VCF file can be converted to several GTrack files, one file per population. In this example run, which uses the atlantic cod data described in section 5.8 we have three different populations. The tool has to be run three times, one for each population.

The first step is to upload the VCF file and (optional) the population files to the current HyperBrowser history. This can be done with the 'Upload File' tool. In the tool user interface, the genome build and the correct VCF file has to be selected. The user has the choice of how to specify the population, either by a population text file, with one individual per line, or as a comma separated list in a text box. The user interface can be seen in figure A.7. When the files are converted, the CSS and FET tools can be run, as described in Appendix A.1.

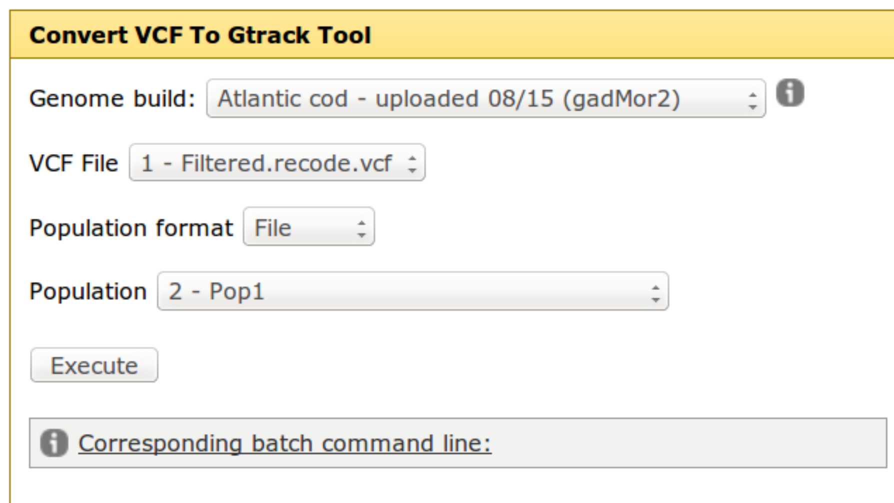


Figure A.7: Convert VCF To GTrack Tool web user interface https://hyperbrowser.uio.no/comparative. This tools takes a VCF file with SNP data and a file of individuals in the population and converts it to a GTrack file. The tool has to be run once for each population.

By following the link given at the start of this section, an example run with the Atlantic cod data, for the marine and coastal 2011 popluations can be found.

Appendix B

Analyses on the Genomic HyperBrowser

The runs for the different analyses can be found at the following Galaxy pages:

- Serial/parallel runtime:
  https://hyperbrowser.uio.no/comparative/u/tuvakt/p/threadtest
- FET three-spined stickleback:
  https://hyperbrowser.uio.no/comparative/u/tuvakt/p/fishers-exact-test
- CSS three-spined stickleback:
  https://hyperbrowser.uio.no/comparative/u/tuvakt/p/cluster-separation-scorer-stickleback
- CSS *Drosophila*:
  https://hyperbrowser.uio.no/comparative/u/tuvakt/p/cluster-separation-scorer-drosophila
- Atlantic cod:
  https://hyperbrowser.uio.no/comparative/u/tuvakt/p/atlantic-cod

For Vederhus' old runs see Vederhus (2013). The results from the *Drosophila* paper (Burke et al., 2010) are included in the *Drosophila* run, and can also be obtained by contacting the authors. The results and data for the stickleback paper (Jones et al., 2012) can be found at the Sticklebrowser, see section 5.6 and A.1.1.

Appendix C

Detailed results

C.1 Detailed results of parallel program

The following tables show the runtimes for the parallel program, with a variable task size and number of threads.

Size of tasks	Total runtime
10	88.9
50	71.6
100	89.3
500	166.5
1000	216.7
5000	306.7
10000	354.8

Table C.1: CSS with a variable task size. The number of threads is 64, and the task size is given in $\{10, 50, 100, 500, 1000, 5000, 10000\}$. The runtime is given in seconds.

Size of tasks	Total runtime
10	103.6
50	29.4
100	18.7
500	14.3
1000	17.2
5000	53.7
10000	93.6

Table C.2: FET with a variable task size. The number of threads is 64, and the task size is given in $\{10, 50, 100, 500, 1000, 5000, 10000\}$. The runtime is given in seconds.

Number of threads	Total runtime
1	734.4
2	374.8
4	194.5
8	119.2
16	96.9
32	89.4
64	88.2
128	87.2

Table C.3: CSS with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 100. The runtime is given in seconds

Number of threads	Speedup
1	1.0
2	1.96
4	3.78
8	6.16
16	7.58
32	8.21
64	8.33
128	8.42

Table C.4: Scaled speedup (scaled for 1 thread) for CSS with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 100. The runtime is given in seconds.

Number of threads	Total runtime
1	1070.2
2	576.8
4	275.4
8	149.5
16	94.1
32	78.1
64	65.5
128	75.5

Table C.5: CSS with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 25. The runtime is given in seconds.

Number of threads	Scaled speedup
1	1.0
2	1.86
4	3.89
8	7.16
16	11.37
32	13.7
64	16.34
128	14.17

Table C.6: Scaled speedup (scaled for 1 thread) for CSS with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 25. The runtime is given in seconds.

Number of threads	Total runtime
1	442.2
2	224.3
4	114.5
8	60.3
16	35.6
32	24.2
64	19.4
128	19.9

Table C.7: FET with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 100. The runtime is given in seconds.

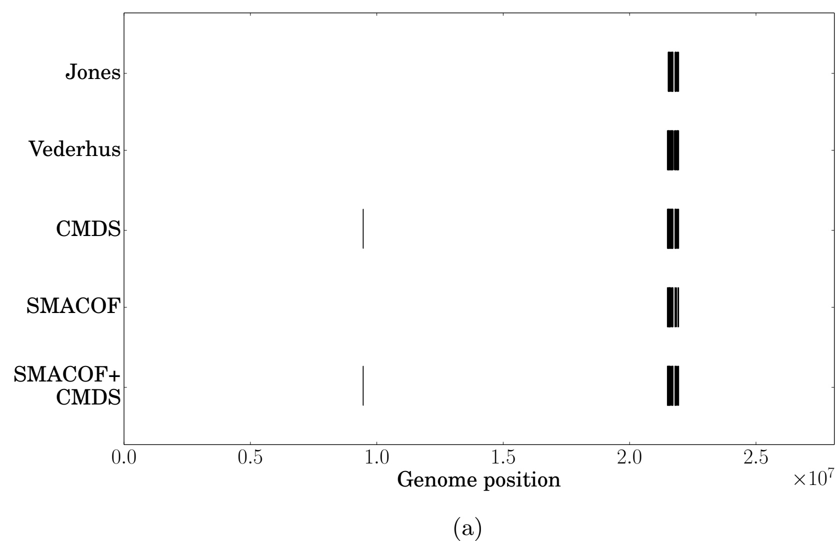
Number of threads	Speedup
1	1.0
2	1.97
4	3.86
8	7.33
16	12.42
32	18.27
64	22.79
128	22.22

Table C.8: Scaled speedup (scaled for 1 thread) for FET with a variable number of threads, given in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The task size is 100. The runtime is given in seconds.

C.2 Detailed three-spined stickleback results

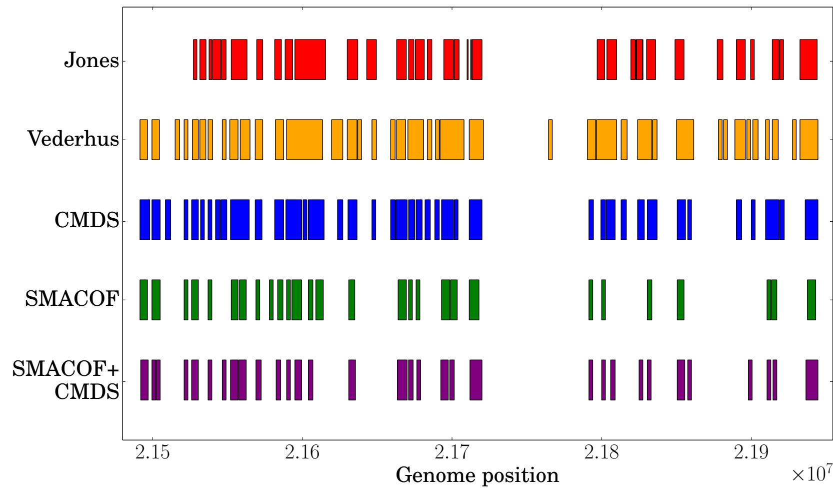
The filtered results for all stickleback chromosomes are included here. Chromosomes without any results and results already included in section 5.6 are not included here.

C.2.1 Cluster Separation Scorer



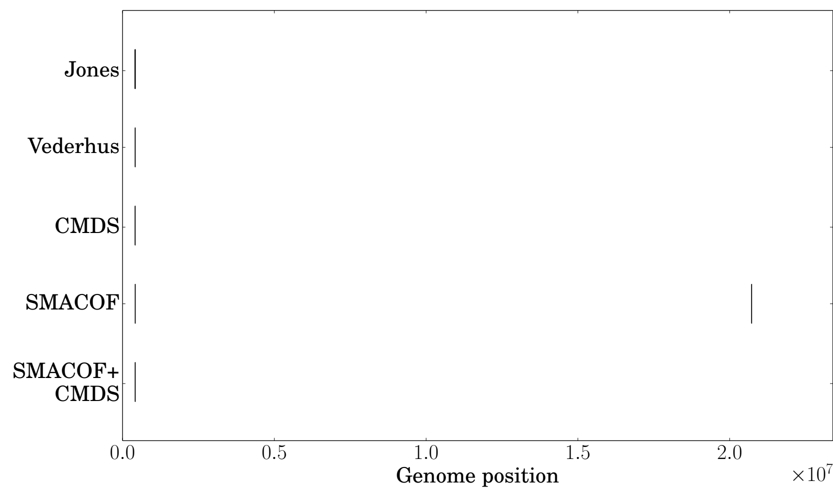
(a)

Figure C.1: Significant regions in stickleback chrI. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.



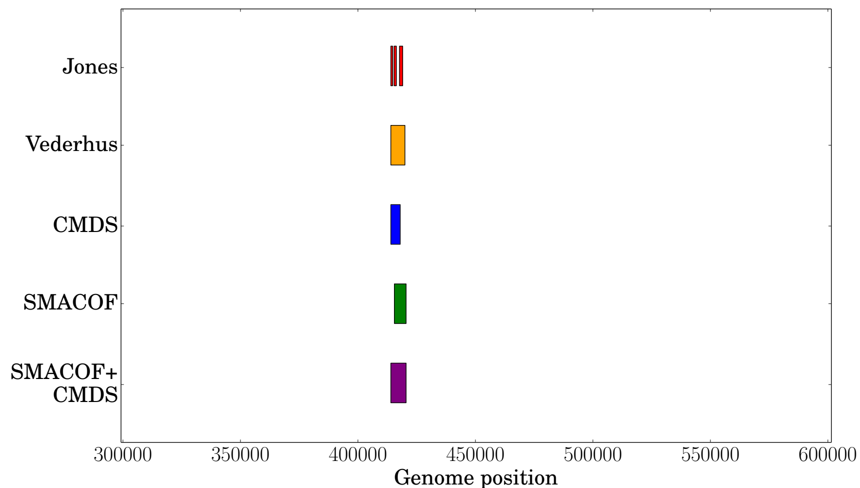
(b) Genome position $[2.15, 2.195] \times 10^7$

Figure C.1: Significant regions in stickleback chrI. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.



(a)

Figure C.2: Significant regions in stickleback chrII. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.



(b) Genome position $[300\,000, 600\,000]$

Figure C.2: Significant regions in stickleback chrII. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.

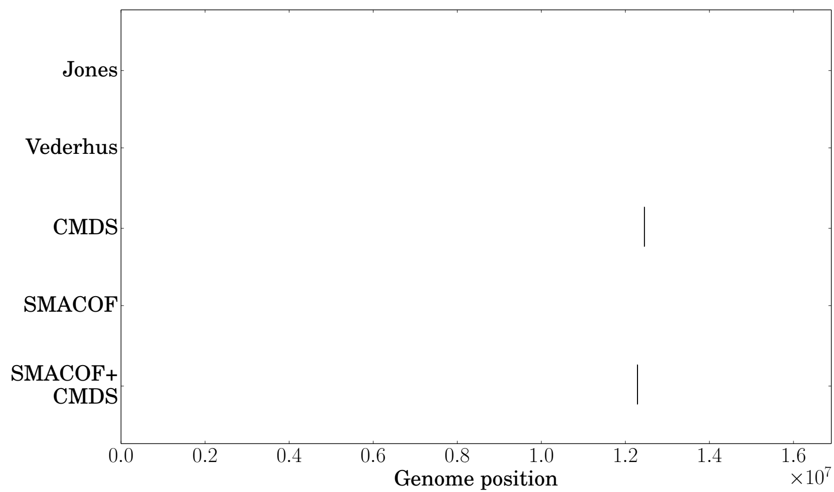
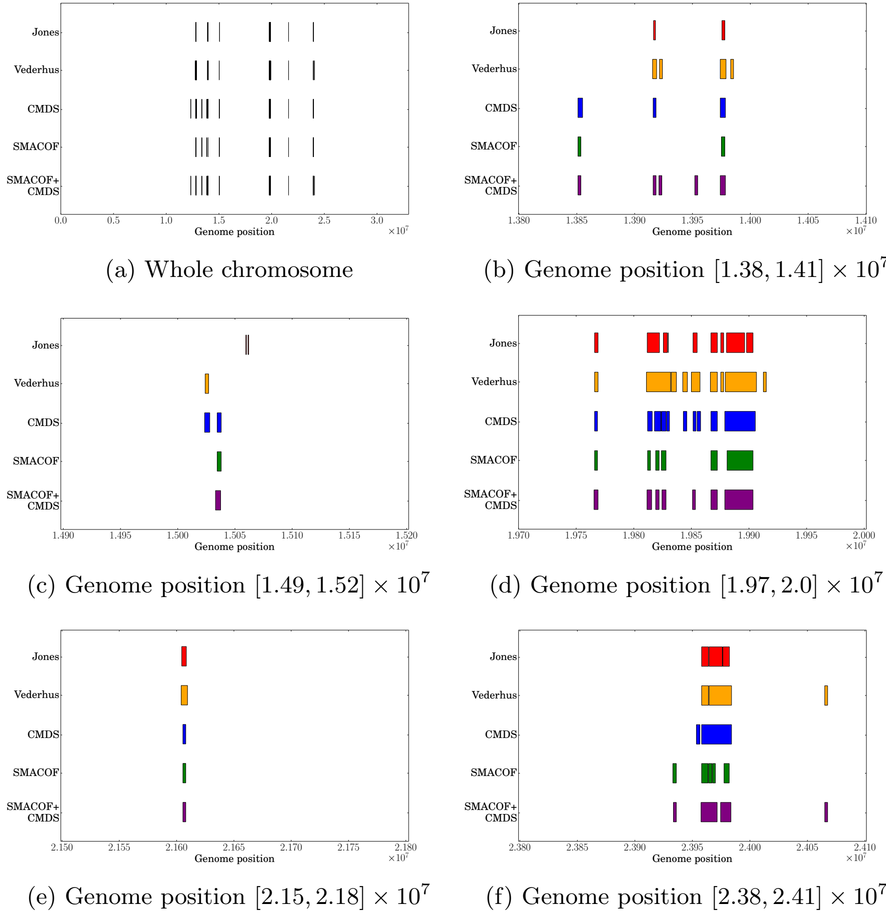


Figure C.3: Significant regions in stickleback chrIII. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.



(a) Whole chromosome

(b) Genome position $[1.38, 1.41] \times 10^7$

(c) Genome position $[1.49, 1.52] \times 10^7$

(d) Genome position $[1.97, 2.0] \times 10^7$

(e) Genome position $[2.15, 2.18] \times 10^7$

(f) Genome position $[2.38, 2.41] \times 10^7$

Figure C.4: Significant regions in stickleback chrIV. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.

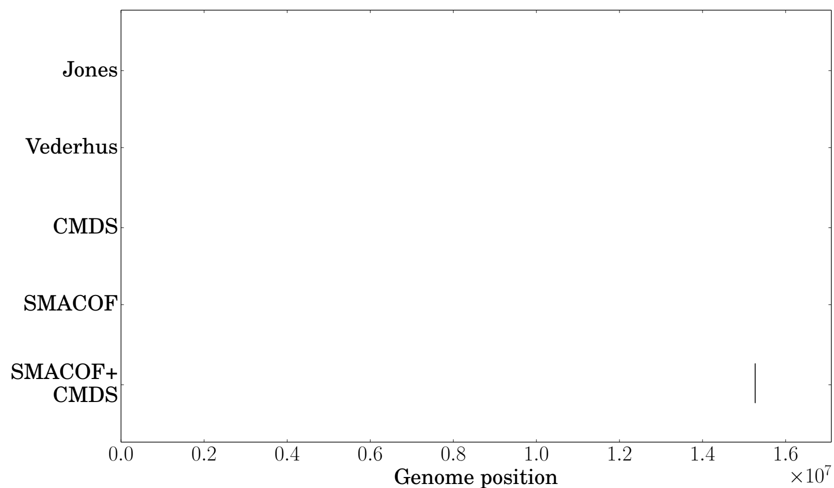
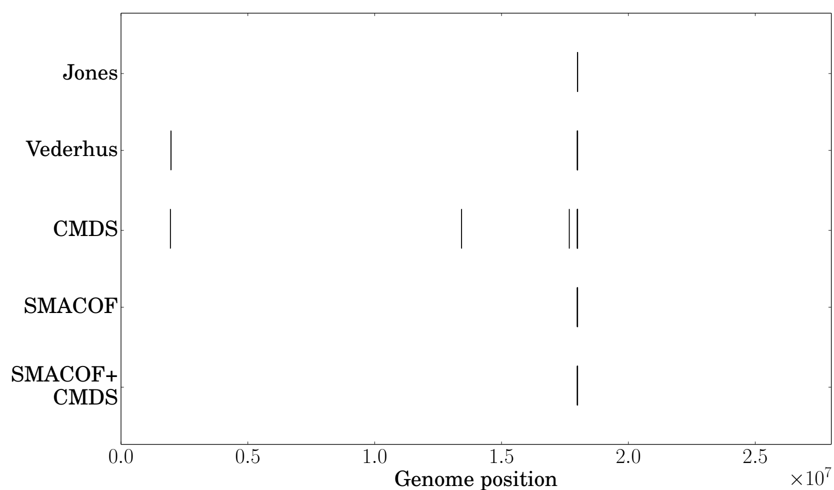
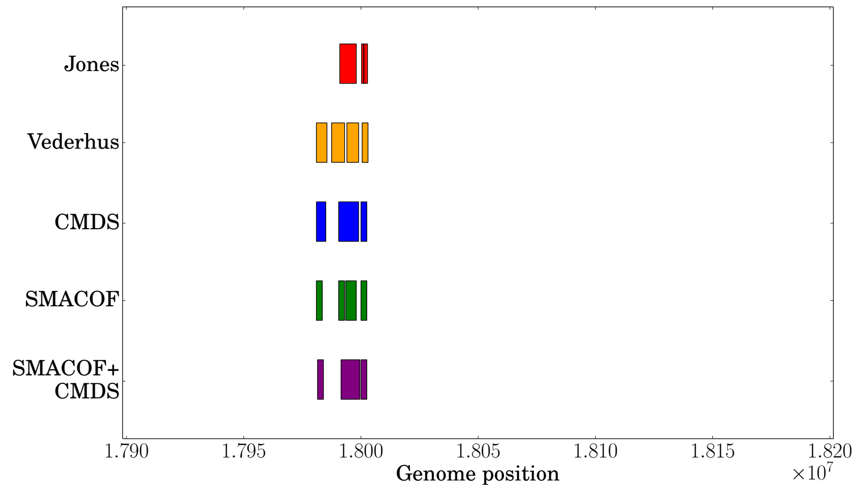


Figure C.5: Significant regions in stickleback chrVI. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.



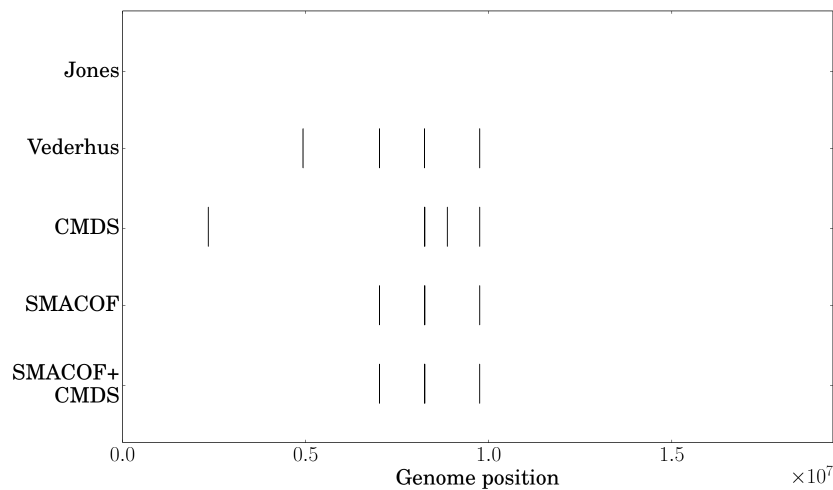
(a)

Figure C.6: Significant regions in stickleback chrVII. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.



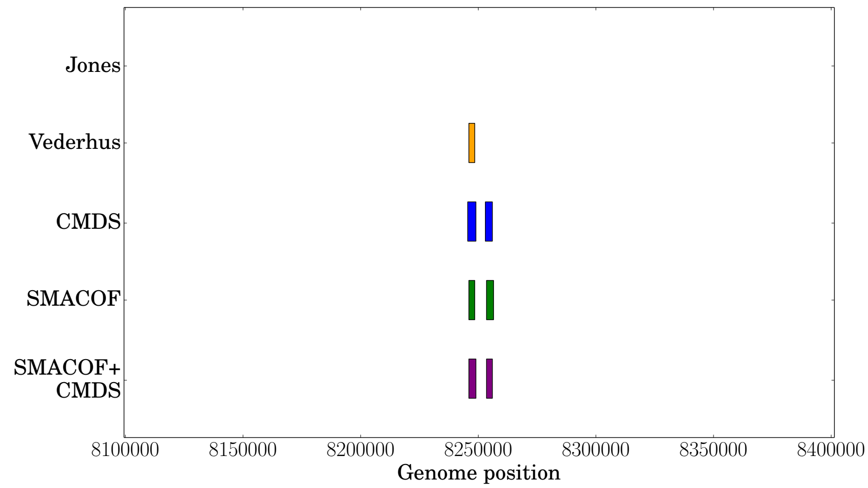
(b) Genome position $[1.79, 1.82] \times 10^7$

Figure C.6: Significant regions in stickleback chrVII. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.

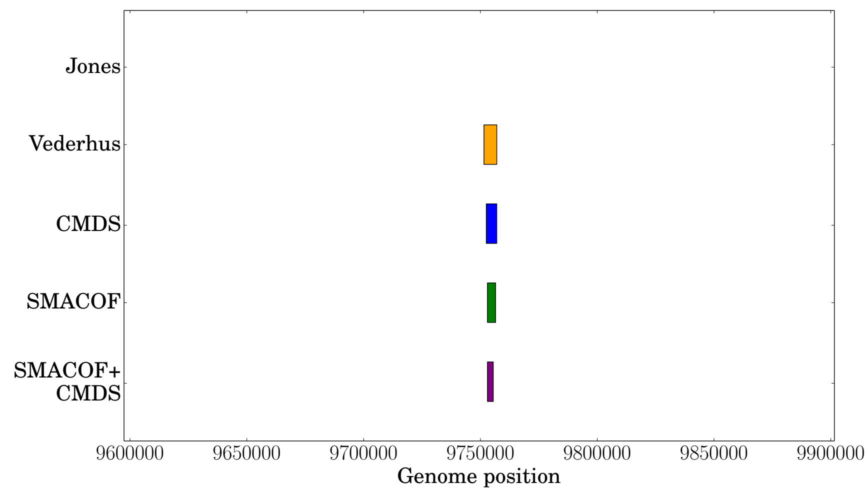


(a)

Figure C.7: Significant regions in stickleback chrVIII. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.



(b) Genome position $[8\,100\,000, 8\,400\,000]$



(c) Genome position $[9\,600\,000, 9\,900\,000]$

Figure C.7: Significant regions in stickleback chrVIII. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.

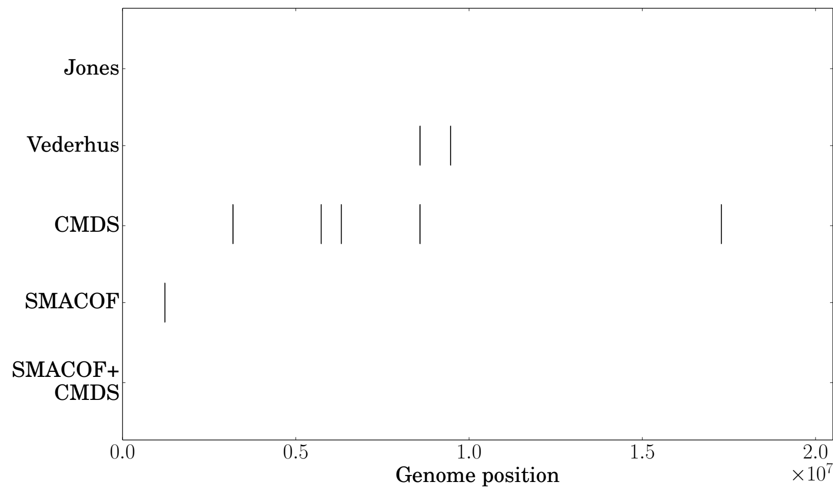


Figure C.8: Significant regions in stickleback chrIX. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.

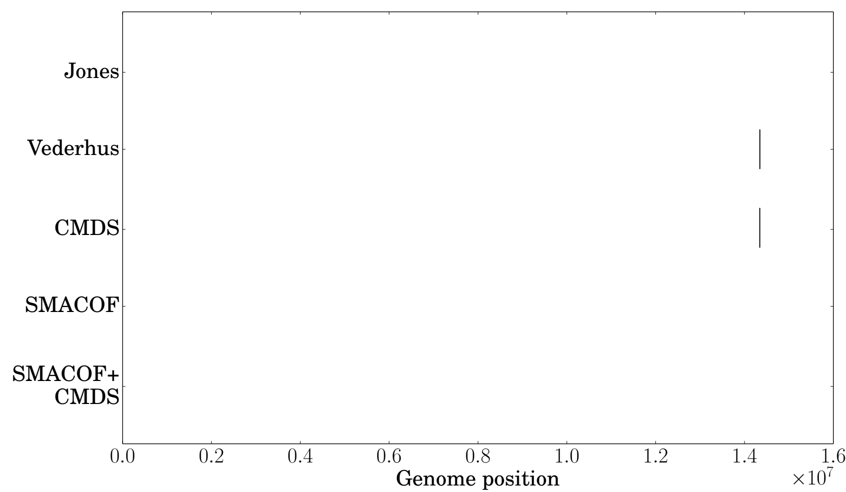
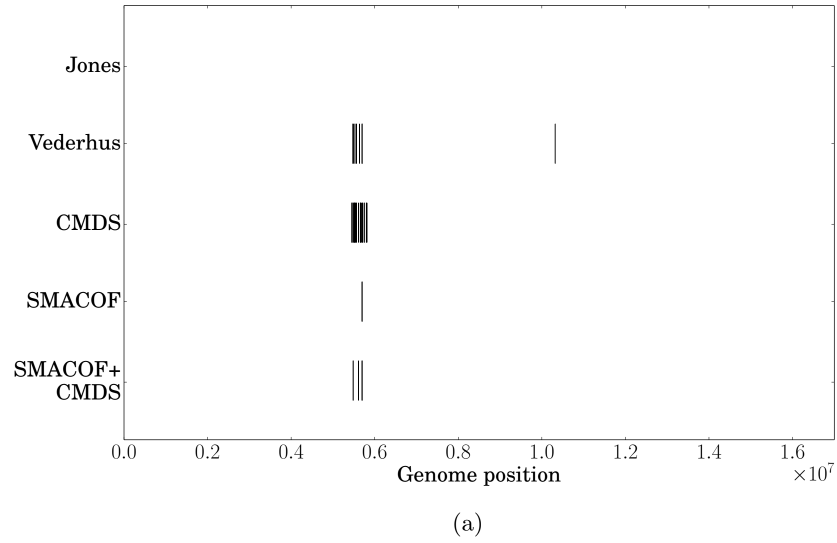
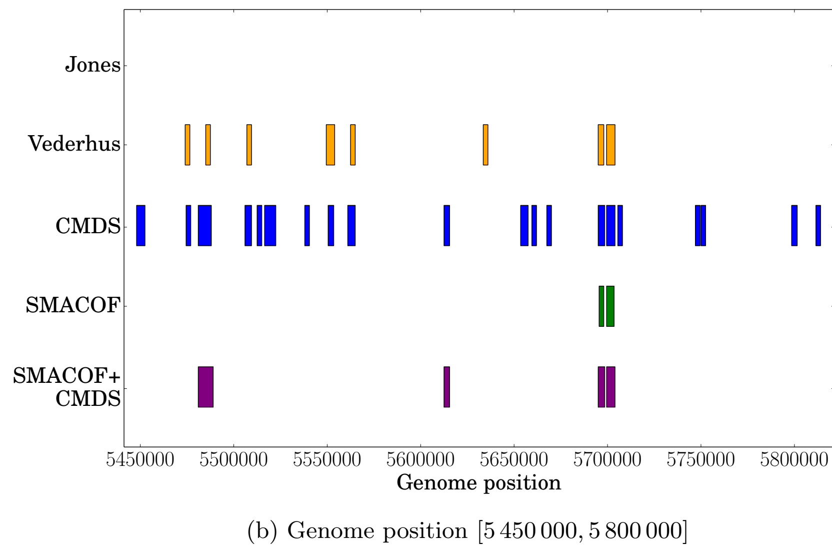


Figure C.9: Significant regions in stickleback chrX. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.



(a)



(b) Genome position $[5\,450\,000, 5\,800\,000]$

Figure C.10: Significant regions in stickleback chrXI. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.

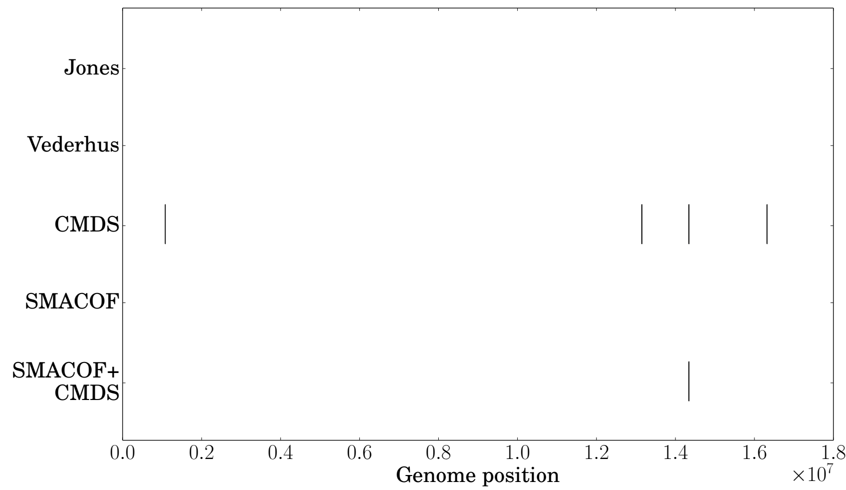


Figure C.11: Significant regions in stickleback chrXII. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.

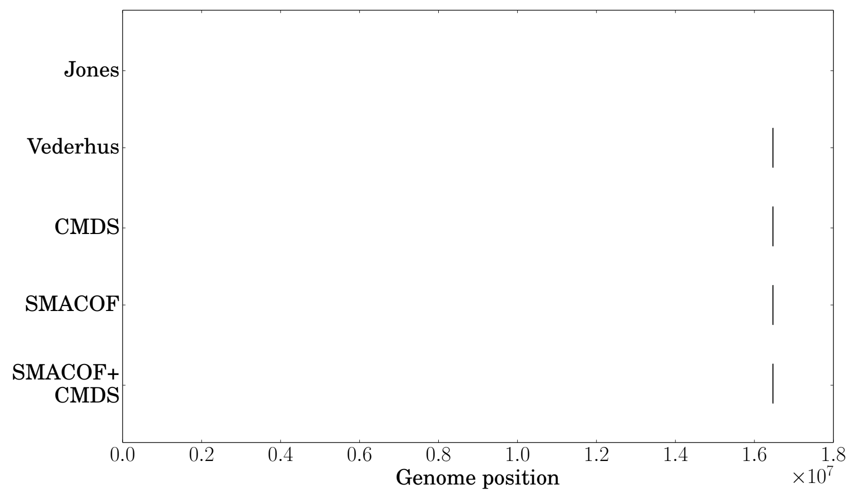
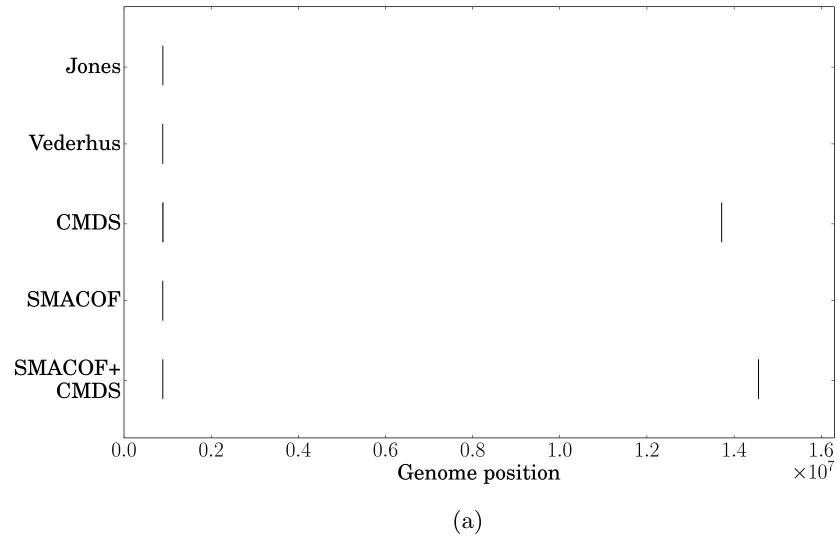
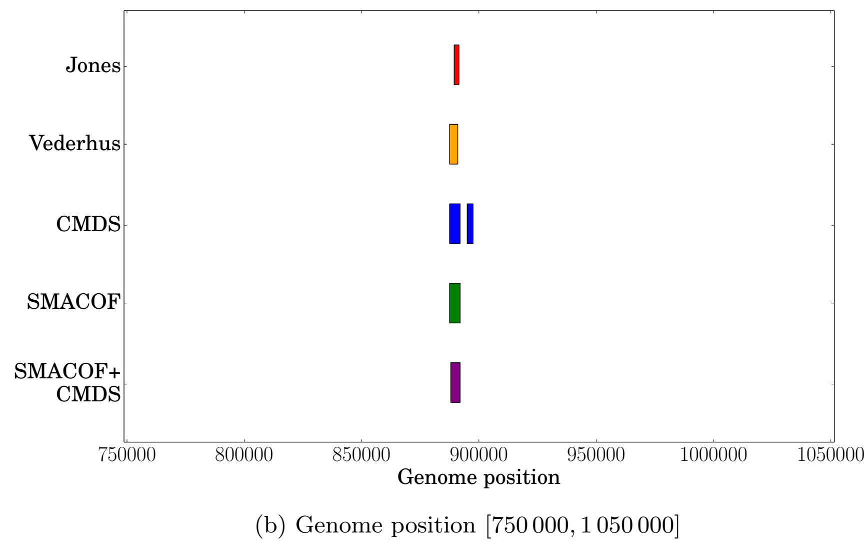


Figure C.12: Significant regions in stickleback chrXVI. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.



(a)



(b) Genome position $[750\,000, 1\,050\,000]$

Figure C.13: Significant regions in stickleback chrXVIII. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.

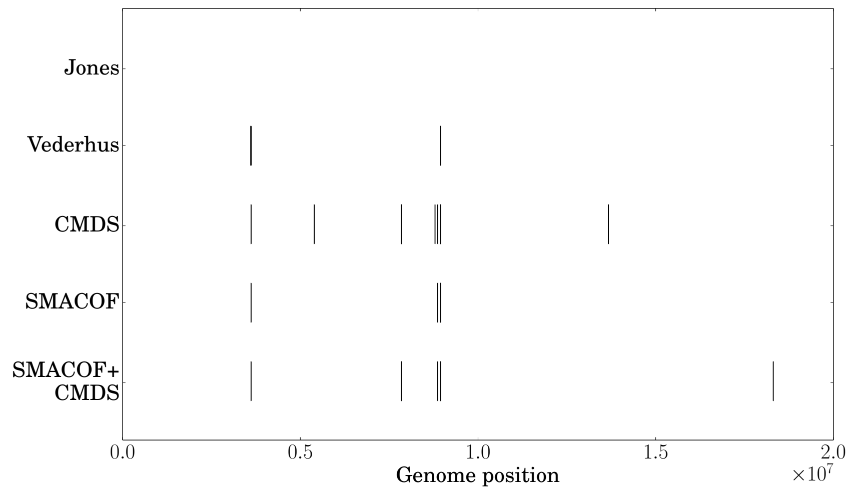


Figure C.14: Significant regions in stickleback chrXX. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.

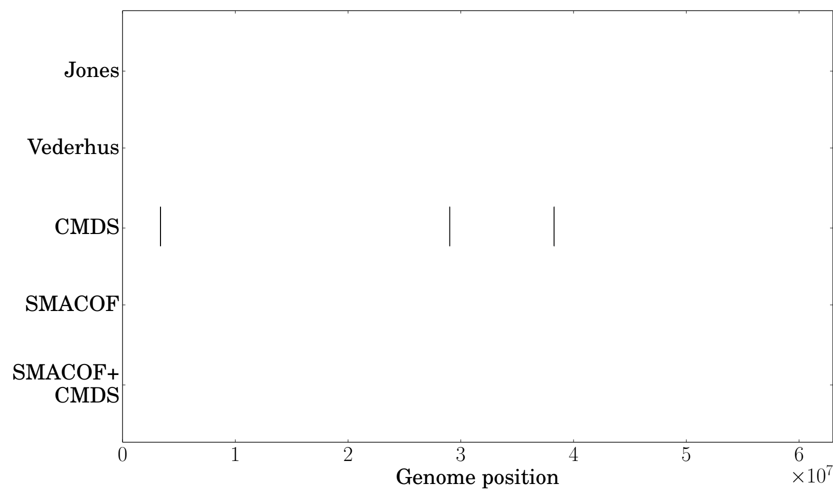
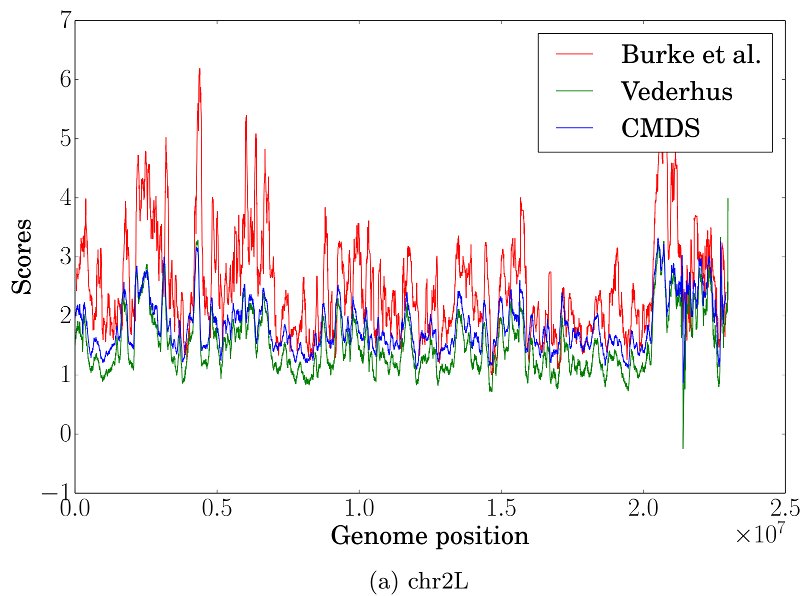


Figure C.15: Significant regions in stickleback chrUn. The strictest results from Jones et al., the FDR 0.05 results from Vederhus and our CSS results from all three MDS methods with FDR 0.05 are shown.

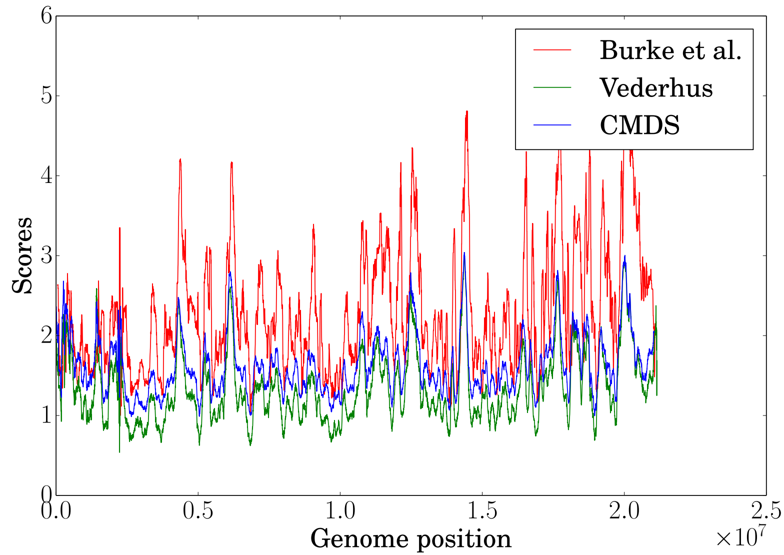
C.3 Detailed *Drosophila* results

All chromosomes, except chromosome X previously shown in section 5.7, are shown here for completeness, for the classical MDS and SMACOF method.

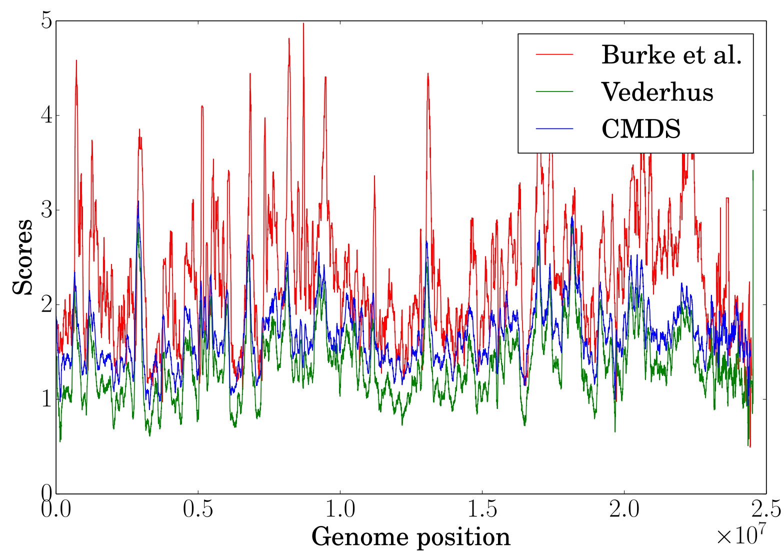


(a) chr2L

Figure C.16: The *Drosophila* results. The Fisher's exact test results from Burke et al., the cluster separation score from Vederhus and our cluster separation score with the classical MDS method are shown. The window and step size for all three analyses are 100kb and 2 kb, respectively.

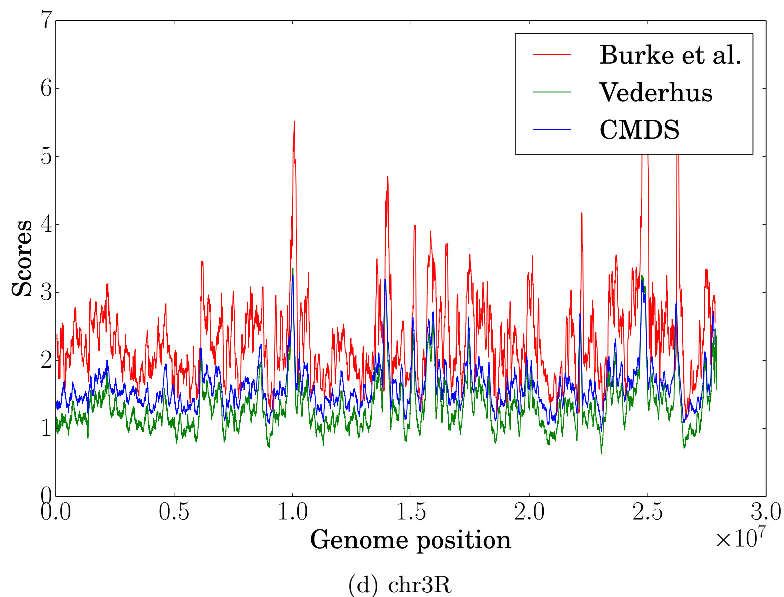


(b) chr2R



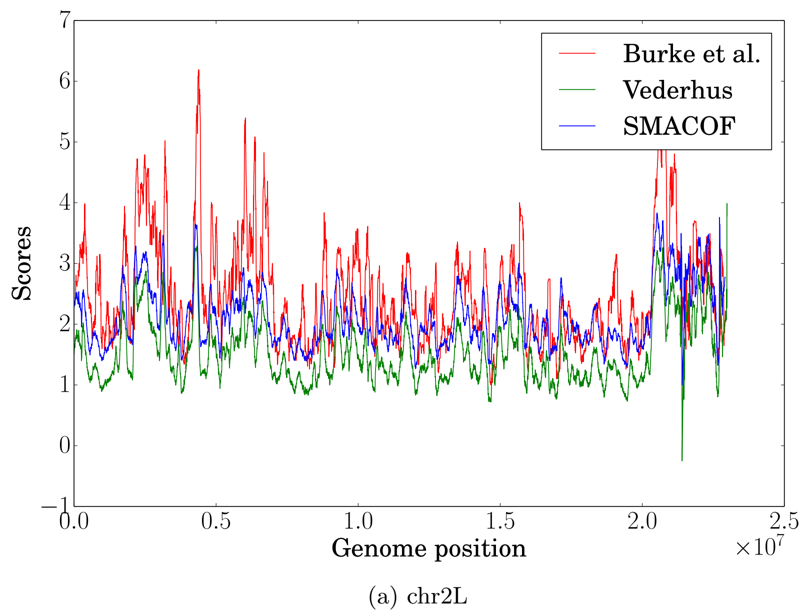
(c) chr3L

Figure C.16: The *Drosophila* results. The Fisher's exact test results from Burke et al., the cluster separation score from Vederhus and our cluster separation score with the classical MDS method are shown. The window and step size for all three analyses are 100kb and 2 kb, respectively.



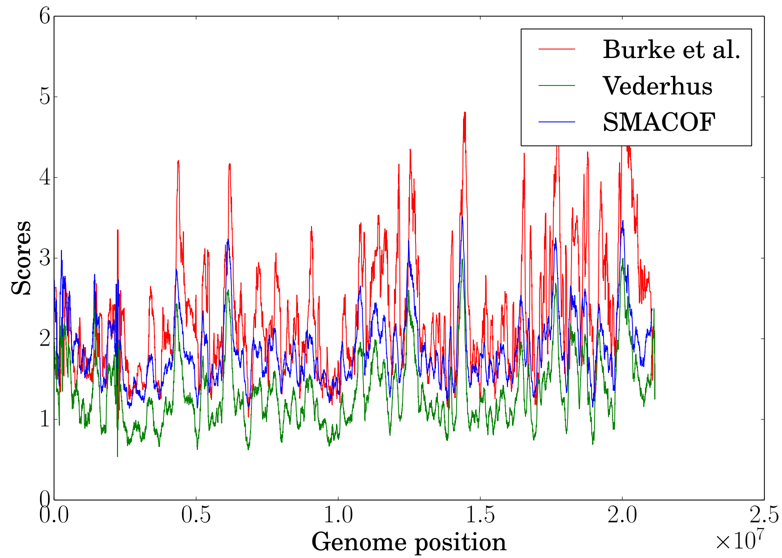
(d) chr3R

Figure C.16: The *Drosophila* results. The Fisher's exact test results from Burke et al., the cluster separation score from Vederhus and our cluster separation score with the classical MDS method are shown. The window and step size for all three analyses are 100kb and 2 kb, respectively.

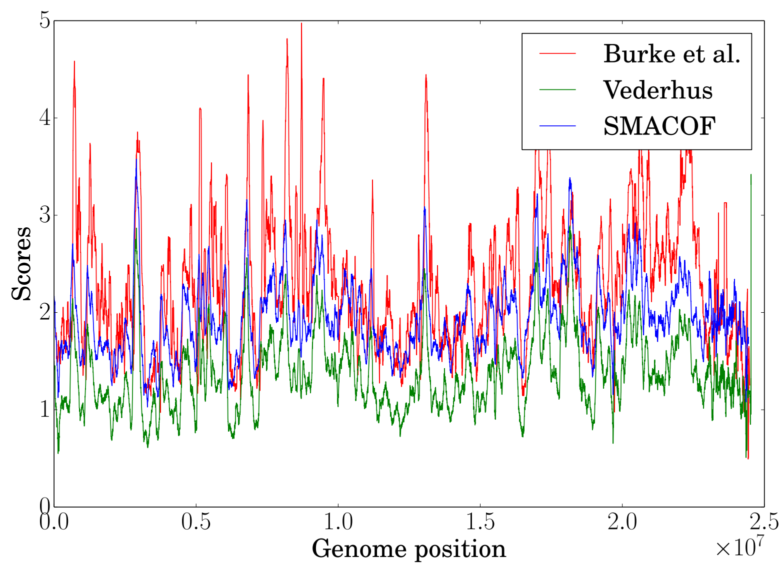


(a) chr2L

Figure C.17: The *Drosophila* results. The Fisher's exact test results from Burke et al., the cluster separation score from Vederhus and our cluster separation score with the SMACOF method for calculating MDS are shown. The window and step size for all three analyses are 100kb and 2 kb, respectively.

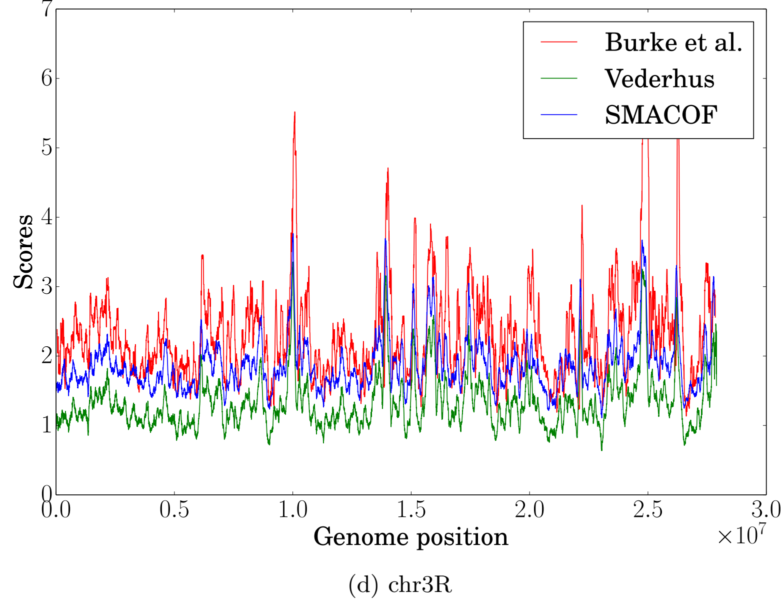


(b) chr2R



(c) chr3L

Figure C.17: The *Drosophila* results. The Fisher's exact test results from Burke et al., the cluster separation score from Vederhus and our cluster separation score with the SMACOF method for calculating MDS are shown. The window and step size for all three analyses are 100kb and 2 kb, respectively.



(d) chr3R

Figure C.17: The *Drosophila* results. The Fisher's exact test results from Burke et al., the cluster separation score from Vederhus and our cluster separation score with the SMACOF method for calculating MDS are shown. The window and step size for all three analyses are 100kb and 2 kb, respectively. As we can see, our results fit the results from Burke et al. well.

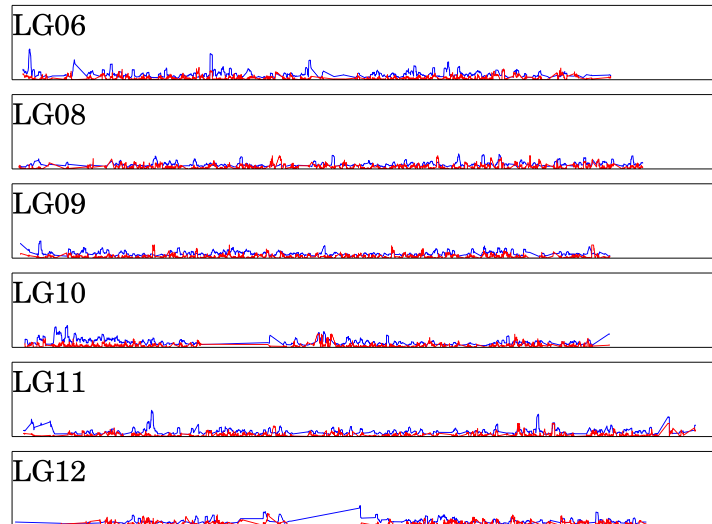
C.4 Detailed Atlantic cod results

C.4.1 The two marine populations

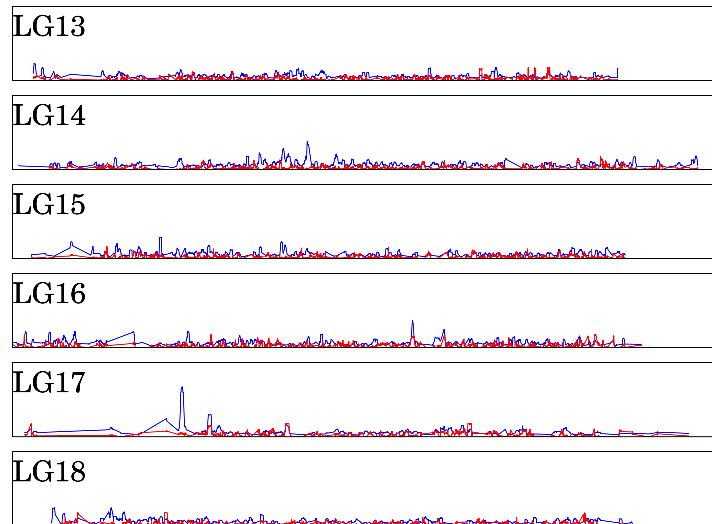
CSS with CMDS results for LG06 and LG08 to LG23 is shown here, for the two marine populations (figure C.17). LG01 to LG05 and LG07 is shown in the result chapter.

C.4.2 The marine and coastal 2011 populations

Filtered regions for the 2011 populations for the CSS can be found in figures C.18 and C.19. The remaining results from the FET can be found in figure C.20.

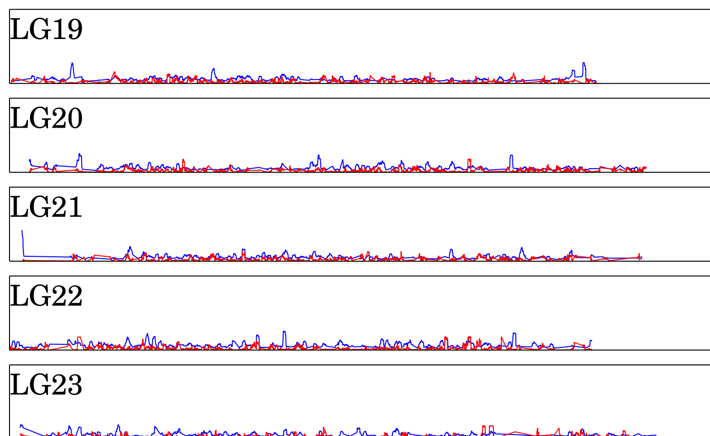


(a) LG06 and LG08 - LG12



(b) LG13 - LG18

Figure C.18: Atlantic cod, marine populations: The results shown are the cluster separation score (CSS) with classical MDS with a window size of 100 kb and a step size of 2 kb. The vertical axis is in $[0, 30]$, and the horisontal axis stretches through the entire chromosome.



(c) LG19 - LG23

Figure C.18: Atlantic cod, marine populations: Cluster separation score (CSS) with classical MDS with a window size of 100 kbp and a step size of 2 kbp. The vertical axis is in $[0, 30]$.

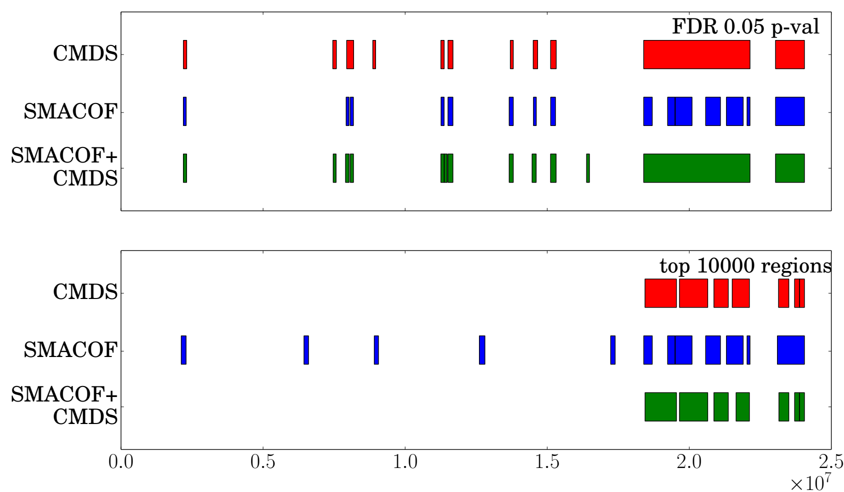


Figure C.19: Atlantic cod, the 2011 populations: filtered cluster separation score (CSS) regions in chromosome LG02. The top figure shows the regions filtered by a FDR p-value of 0.05, while the bottom shows the top 10 000 regions filtered by CSS score. We see the results for the classical MDS, SMACOF and combination method (SMACOF+CMDS) for calculating MDS.

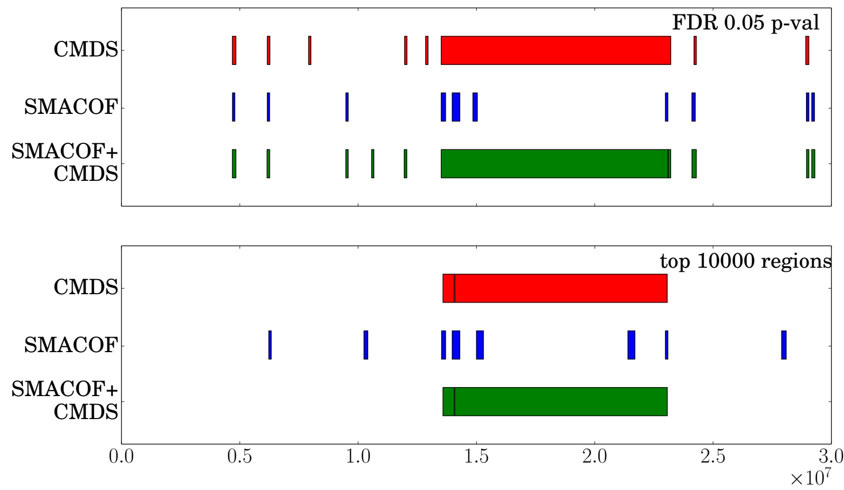
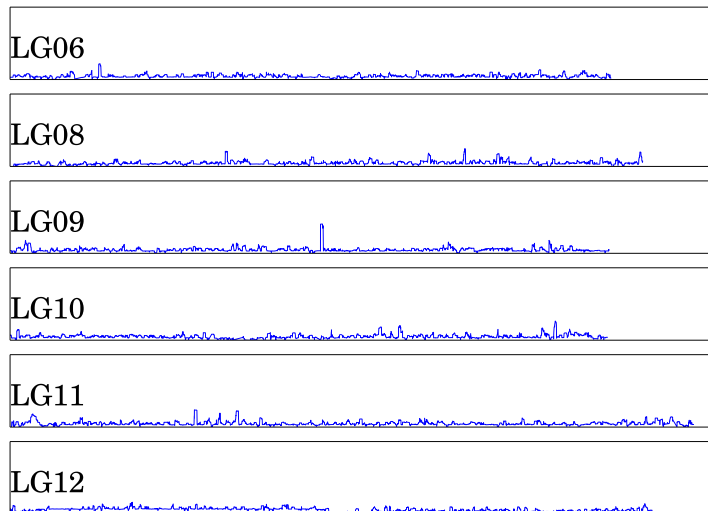
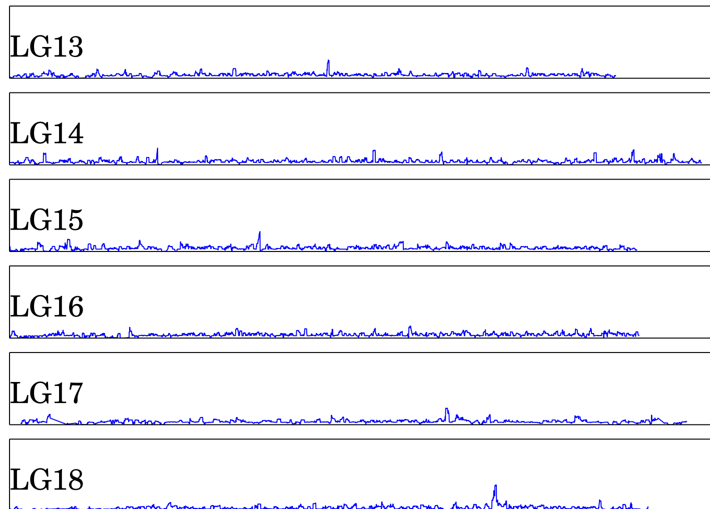


Figure C.20: Atlantic cod, the 2011 populations: filtered cluster separation score (CSS) regions in chromosome LG07. The top figure shows the regions filtered by a FDR p-value of 0.05, while the bottom shows the top 10 000 regions filtered by CSS score. We see the results for the classical MDS, SMACOF and combination method (SMACOF+CMDS) for calculating MDS.

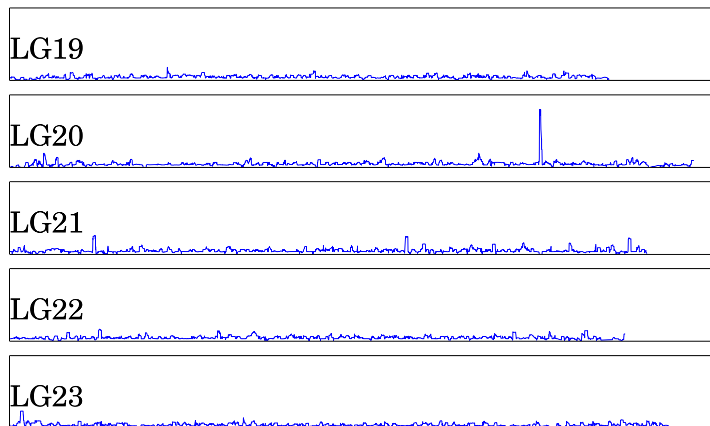


(a) LG06 and LG08 - LG12

Figure C.21: Atlantic cod, the 2011 populations: Fisher's exact test (FET) with a window size of 100 kb and a step size of 2 kb. The vertical axis is in $[0, 10]$.



(b) LG13 - LG18



(c) LG19 - LG23

Figure C.21: Atlantic cod, the 2011 populations: Fisher's exact test (FET) with a window size of 100 kb and a step size of 2 kb. The vertical axis is in $[0, 10]$.

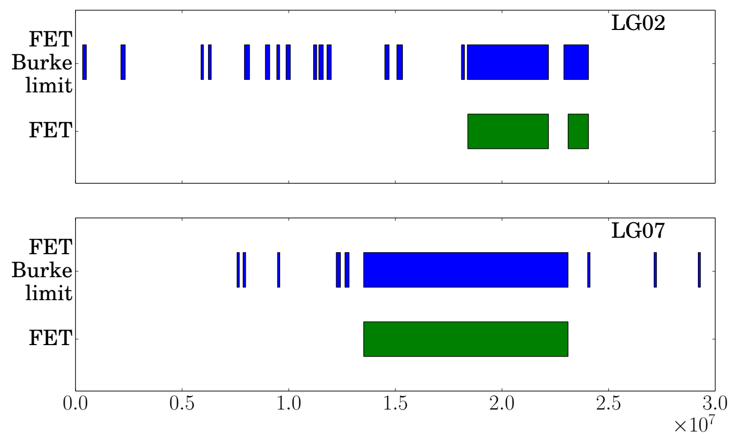


Figure C.22: Atlantic cod, the 2011 populations: filtered Fisher's exact test (FET) regions in chromosomes LG02 and LG07 for a 100 kb FET. The results from the Burke et al. limit and the strict limit given in section 5.8.2 are shown. The top figure shows LG02 and the bottom figure shows LG07.

Appendix D

Source code

The source code can be found at:
http://tuvakt.github.io/Fast-Parallel-Tools-for-Genome-wide-Analysis-of-Genomic-Divergence

References

Alberts, B., Bray, D., Hopkin, K., Johnson, A., Lewis, J., Raff, M., ... Walter, P. (2014a). Fron DNA to Protein: How Cells Read the Genome. In *Essential Cell Biology* (4th, Chap. 7, pp. 223–260). New York: Garland Science.

Alberts, B., Bray, D., Hopkin, K., Johnson, A., Lewis, J., Raff, M., ... Walter, P. (2014b). How Genes and Genomes Evolve. In *Essential Cell Biology* (4th, Chap. 9, pp. 289–324). New York: Garland Science.

Baldwin-Brown, J. G., Long, A. D. & Thornton, K. R. (2014). The Power to Detect Quantitative Trait Loci Using Resequenced, Experimentally Evolved Populations of Diploid, Sexual Organisms. *Molecular Biology and Evolution*, *31*(4), 1040–1055. doi:10.1093/molbev/msu048

Behnel, S., Bradshaw, R. & Seljebotn, D. (2009). Cython tutorial. In G. Varoquaux, S. van der Walt & J. Millman (Eds.), *Proceedings of the 8th Python in Science Conference* (pp. 4–14). Pasadena, CA USA. Retrieved from http://conference.scipy.org/proceedings/SciPy2009/paper_1

Blankenberg, D., Kuster, G. V., Coraor, N., Ananda, G., Lazarus, R., Mangan, M., ... Taylor, J. (2001). Galaxy: A Web-Based Genome Analysis Tool for Experimentalists. In *Current Protocols in Molecular Biology*. John Wiley & Sons, Inc. doi:10.1002/0471142727.mb1910s89

Borg, I. & Groenen, P. J. F. (2005). A Majorization Algorithm for Solving MDS. In *Modern Multidimensional Scaling* (pp. 169–197). Springer Series in Statistics. New York: Springer. doi:10.1007/0-387-28981-X_8

Borg, I., Groenen, P. J. F. & Mair, P. (2013). MDS Algorithms. In *Applied Multidimensional Scaling* (pp. 81–86). SpringerBriefs in Statistics. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-31848-1_8

Burke, M. K., Dunham, J. P., Shahrestani, P., Thornton, K. R., Rose, M. R. & Long, A. D. (2010). Genome-wide analysis of a long-term evolution experiment with Drosophilia. *Nature*, *467*, 587–590. doi:10.1038/nature09352

Burke, M. K., Liti, G. & Long, A. D. (2014). Standing Genetic Variation Drives Repeatable Experimental Evolution in Outcrossing Populations of Saccharomyces cerevisiae. *Molecular Biology and Evolution*, *31*(12), 3228–3239. doi:10.1093/molbev/msu256

Colosimo, P. F., Hosemann, K. E., Balabhadra, S., Villarreal, G., Dickson, M., Grimwood, J., ... Kingsley, D. M. (2005). Widespread Parallel Evolution in Sticklebacks by Repeated Fixation of Ectodysplasin Alleles. *Science*, *307*(5717), 1928–1933. doi:10.1126/science.1107239

Crans, G. G. & Shuster, J. J. (2008). How conservative is Fisher's exact test? A quantitative evaluation of the two-sample comparative binomial trial. *Statistics in Medicine*, *27*(18), 3598–3611. doi:10.1002/sim.3221

Crick, F. (1958). On Protein Synthesis. *The Symposia of the Society for Experimental Biology*, *12*, 138–163. Retrieved August 9, 2015, from http://profiles.nlm.nih.gov/ps/retrieve/ResourceMetadata/SCBBZY

Crick, F. (1970). Central Dogma of Molecular Biology. *Nature*, *227*, 561–563. doi:doi:10.1038/227561a0

Danecek, P., Auton, A., Abecasis, G., Albers, C. A., Banks, E., DePristo, M. A., ... Group, 1. G. P. A. (2011). The variant call format and VCFtools. *Bioinformatics*, *27*, 2156–2158. doi:10.1093/bioinformatics/btr330

De Leeuw, J. (1977). Applications of convex analysis to multidimensional scaling. In J. Barra, F. Brodeau, G. R. Romier & B. van Cutsem (Eds.), *Recent developments in statistics* (pp. 133–146). Amsterdam: North Holland Publishing Company. Retrieved August 9, 2015, from https://escholarship.org/uc/item/7wg0k7xq

De Leeuw, J. (1988). Convergence of the majorization method for multidimensional scaling. *Journal of Classification*, *5*(2), 163–180. doi:10.1007/BF01897162

Dean, J. & Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, *51*(1), 107–113. doi:10.1145/1327452.1327492

Durstenfeld, R. (1964). Algorithm 235: Random Permutation. *Commun. ACM*, *7*(7), 420–. doi:10.1145/364520.364540

Feldman, S. E. & Klinger, E. (1963). Short cut calculation of the Fisher-Yates "exact test". *Psychometrika*, *28*(3), 289–291. doi:10.1007/BF02289576

Galassi, M. et al. (2009). *GNU Scientific Library Reference Manual*. 3rd. Retrieved from http://www.gnu.org/software/gsl/

Giardine, B., Riemer, C., Hardison, R. C., Burhans, R., Elnitski, L., Shah, P., ... Nekrutenko, A. (2005). Galaxy: A platform for interactive large-scale genome analysis. *Genome Research*, *15*(10), 1451–1455. doi:10.1101/gr.4086505

Goecks, J., Nekrutenko, A. & Taylor, J. (2010). Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, *11*(8). doi:10.1186/gb-2010-11-8-r86

Graham, S. L., Kessler, P. B. & Mckusick, M. K. (1982). Gprof: A Call Graph Execution Profiler. *SIGPLAN Not. 17*(6), 120–126. doi:10.1145/872726.806987

Grama, A., Gupta, A., Karypis, G. & Kumar, V. (2003). *Introduction to parallel computing* (2nd). Harlow: Pearson Education Limited.

Gundersen, S., Kalaš, M., Abul, O., Frigessi, A., Hovig, E. & Sandve, G. K. (2011). Identifying elemental genomic track types and representing them uniformly. *BMC Bioinformatics*, *12*(1), 1–17. doi:10.1186/1471-2105-12-494

Jones, F. C., Grabherr, M. G., Chan, Y. F., Russell, P., Mauceli, E., Johnson, J., . . . Kingsley, D. M. (2012). The genomic basis of adaptive evolution in threespine sticklebacks. *Nature*, *484*, 55–61. doi:10.1038/nature10944

Karolchik, D., Hinrichs, A. S., Furey, T. S., Roskin, K. M., Sugnet, C. W., Haussler, D. & Kent, W. J. (2004). The UCSC Table Browser data retrieval tool. *Nucleic Acids Research*, *32*(suppl 1), D493–D496. doi:10.1093/nar/gkh103

Kernighan, B. W. & Ritchie, D. M. (1988). *The C programming language* (2nd). Englewood Cliffs, N.J: Prentice Hall.

Klug, W. S., Cummings, M. R. & Spencer, C. (2007). Introduction to Genetics. In *Essentials of Genetics* (Chap. 1, pp. 1–16). San Fransisco: Pearson/-Prentice Hall.

Knuth, D. E. (1974). Structured Programming with Go to Statements. *ACM Comput. Surv. 6*(4), 261–301. doi:10.1145/356635.356640

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (3rd). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Kruskal, J. B. (1964). Nonmetric multidimensional scaling: A numerical method. *Psychometrika*, *29*(2), 115–129. doi:10.1007/BF02289694

Langtangen, H. P. (2009). *Python Scripting for Computational Science* (3rd). Berlin: Springer.

Lillesæther, J. L. (2011). *Retroactively Parallelizing a Large Python System* (Master's thesis, University of Oslo). Retrieved from http://urn.nb.no/URN:NBN:no-28918

Marsland, S. (2009). Dimensionality Reduction. In *Machine learning: an algorithmic perspective* (1st, Chap. 10, pp. 221–246). Boca Raton, Fla: Chapman & Hall/CRC.

McDonald, J. H. (2014). *Handbook of biological statistics* (3rd). Baltimore, Maryland: Sparky House Publishing.

Nichols, B., Buttlar, D. & Farrell, J. P. (1996). *Pthreads programming* (1st). Bonn: O'Reilly & Associtates, Inc.

North, B. V., Curtis, D. & Sham, P. C. (2002). A Note on the Calculation of Empirical P Values from Monte Carlo Procedures. *American Journal of Human Genetics*, *71*, 439–441. doi:10.1086/341527

Phipson, B. & Smyth, G. K. (2010). Permutation P-values should never be zero: calculating exact P-values when permutations are randomly drawn. *Statistical Applications in Genetics and Molecular Biology*, *9*(1). doi:10.2202/1544-6115.1585

Quinn, M. J. (2003). *Parallel programming in C with MPI and OpenMP*. New York: McGraw-Hill Education Group.

Rognes, T., Mahé, F. & xflouris. (2015). Vsearch: VSEARCH version 1.0.16. doi:10.5281/zenodo.15524

Roweis, S. T. & Saul, L. K. (2000). Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, *290*(5500), 2323–2326. doi:10.1126/science.290.5500.2323

Sandve, G. K., Gundersen, S., Johansen, M., Glad, I. K., Gunathasan, K., Holden, L., ... Hovig, E. (2013). The Genomic HyperBrowser: an analysis web server for genome-scale data. *Nucleic Acids Research*, *41*(W1), W133–W141. doi:10.1093/nar/gkt342

Sandve, G. K., Gundersen, S., Rydbeck, H., Glad, I. K., Holden, L., Holden, M., ... Hovig, E. (2010). The Genomic HyperBrowser: inferential genomics at the sequence level. *Genome Biology*, *11*(12), R121. doi:10.1186/gb-2010-11-12-r121

Seward, J. & Nethercote, N. (2005). Using Valgrind to Detect Undefined Value Errors with Bit-precision. In *Proceedings of the annual conference on usenix annual technical conference* (pp. 2–2). ATEC '05. Anaheim, CA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=1247360.1247362

Sommerville, I. (2011). Agile software development. In *Software engineering* (9th, pp. 56–81). Boston, Mass: Pearson.

Tenenbaum, J. B., de Silva, V. & Langford, J. C. (2000). A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, *290*(5500), 2319–2323. doi:10.1126/science.290.5500.2319

Tzeng, J., Lu, H. H.-S. & Li, W.-H. (2008). Multidimensional scaling for large genomic data sets. *BMC Bioinformatics*, *9*, 179. doi:doi:10.1186/1471-2105-9-179

Vederhus, T. (2013). *Tools for Genome-wide Analysis of Genomic Divergence* (Master's thesis, University of Oslo). Retrieved from http://urn.nb.no/URN:NBN:no-39032

Zar, J. H. (1987). A fast and efficient algorithm for the Fisher exact test. *Behavior Research Methods, Instruments & Computers*, *19*(4), 413–414. doi:10.3758/BF03202590